University of Alberta

Flexible tool support for Software Product Lines

by

Iliyan G. Kaytazov

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science

Department of Electrical and Computer Engineering

Edmonton, Alberta
Fall 2002

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-81418-1

Canada

University of Alberta

Library Release Form

Name of Author: Iliyan Kaytazov

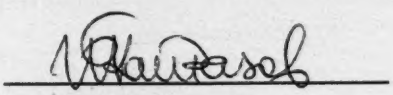
Title of Thesis: Flexible tool support for Software Product Lines

Degree: Master of Science

Year this Degree Granted: 2002

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.



Iliyan Kaytazov,
11504 – 78 Ave, Edmonton
Alberta, Canada
T6G 0N5

Date: Aug. 29. 2002

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled Flexible tool support for Software Product Lines submitted by Iliyan Kaytazov in partial fulfillment of the requirements for the degree of Master of Science.

Supervisor: Dr. Petr Musilek

Dr. Marek Reformat

Dr. Eleni Stroulia

Date: Aug 26, 2002

Abstract

Software Product Lines (SPL) are a promising approach to develop multiple products by providing a reduction in rework and a systematic way to exploit the synergistic relationships between products. One of the serious obstacles in implementing an SPL approach to developing software is its complexity and the large number of activities over which it spans. Consequently, a tool that alleviates the difficulty of adopting an SPL approach is required. Such a tool should be designed and implemented with a few considerations in mind. Given the diversity of SPL activities, the possibility for easy integration of third-party tools is essential. Furthermore, the tool should be designed in such a way that it can be easily modified. The proposed design for and implementation of such a tool covers these requirements by leveraging existing knowledge in the field of software architecture, design patterns, and emerging object-oriented technologies.

Acknowledgment

First, I would like to thank Dr. Petr Musilek for his help, advice, and encouragement during the last year of my research. Without his support, and valuable comments this work would have not been concluded.

Also, I wish to express my gratitude to Dr. Giancarlo Succi under whose supervision this work originally started. I appreciate his criticism, guidance, and inspiration. Thanks also go to Jason Yip, for his helpful advice and insights; to Rohit Gupta and JR Lyon for their contribution to this work.

Finally, I would like to thank my parents, Georgy and Lily, for their warmth, true love, and continued support for any scholastic endeavor of mine; and my brother, Vladimir, for his positive influence and for instilling in me avidity for reading since my earliest age.

Table of Contents

1 INTRODUCTION 1

2 RELATED WORK 4

2.1 RSEB 4

2.1.1 Obstacles in adopting large-scale reuse 4

2.1.2 Changes in the process 6

2.1.3 Changes in the organization 8

2.1.4 Object-Oriented System Modelling 11

2.1.5 Application Family Development 12

2.2 PuLSE 14

2.2.1 Shortcomings of domain engineering methodologies 14

2.2.2 Overview 15

2.2.3 Infrastructure Construction 17

2.2.4 Usage 20

2.2.5 Evolution 21

2.2.6 Support 21

2.3 FAST 23

2.3.1 Careful Engineering vs. Rapid Production 24

2.3.2 Principles of Family-based development 25

2.3.3 Overview of FAST 27

2.3.4 Process model of FAST 30

2.4 Summary 33

3 SHERLOCK – A METHODOLOGY FOR ADOPTING SPL DEVELOPMENT 35

3.1 Domain Definition 37

3.2 Domain Characterization 39

3.3 Domain Scoping 42

3.4 Domain Modelling 44

3.5 Domain Framework Development 46

3.6 SUMMARY 50

4 NEED AND REQUIREMENTS FOR AN SPL SUPPORT TOOL 52

5 HOLMES – A SUPPORT TOOL FOR SPL DEVELOPMENT 57

5.1 BLACKBOARD SYSTEMS 57

5.2 TUPLE SPACE 63

5.3 PERSISTENT STORAGE 65

5.4 CRITIQUING SYSTEM 66

5.4.1 Rationale 67

5.4.2 Holmes' Implementation 69

5.4.3 Problems encountered while using JPL. 75

5.5 SAMPLE SESSION 76

5.6 SUMMARY 85

6 CASE STUDY 87

6.1 DOMAIN DEFINITION 87

6.1.1 Information about the domain 87

6.1.2 Domain Vocabulary 90

6.1.3 Information about the market 90

6.1.4 Overall strategy 91

6.1.5 Definition of domains 91

6.1.6 Feasibility analysis 92

6.2 DOMAIN CHARACTERIZATION 92

6.2.1 Simulation software 92

6.2.2 Embedded software 94

6.2.3 Market segments and users 94

6.3 DOMAIN SCOPING 95

6.3.1 Variation points and attributes 95

6.3.2 Product strategies 97

6.4 DOMAIN MODELLING 99

6.4.1 Scenarios 99

6.4.2 Analysis Model 100

6.4.3 Design Model *101*

6.5 SUMMARY 103

7 CONCLUSION AND FUTURE WORK **105**

8 REFERENCES **108**

List of Tables

Table 1: Sherlock Phases 36
Table 2: Mapping of existing tools to identified requirements 56
Table 3: Critique Engine Initialization 74
Table 4: Mapping to requirements 85
Table 5: Domains 92
Table 6: Classes and Responsibilities 101

List of Figures

Figure 1: Organization Structure (Jacobson et al., 1997) 8
Figure 2: Incremental Adoption of Reuse (Jacobson et al., 1997) 9
Figure 3: PuLSE Structure (Bayer et al., 1999) 16
Figure 4: FAST sub-processes (Weiss and Lai, 1999) 23
Figure 5: Economic Model (Weiss and Lai, 1999) 28
Figure 6: The FAST hierarchy 32
Figure 7: Use Case for PL support tool 53
Figure 8: Architecture 57
Figure 9: Critiquing System Architecture 70
Figure 10: Configuring the Critiquing System 70
Figure 11: Class Hierarchy 72
Figure 12: Example of a rule definition in JPL vs. Prolog 73
Figure 13: Compound Structure 76
Figure 14: Domain vocabulary 78
Figure 15: Domain description 78
Figure 16: Classified information 79
Figure 17: Feasibility Analysis 79
Figure 18: Diagrams of value 80
Figure 19: Compatibility types 81
Figure 20: User Flows 82
Figure 21: Domain Scoping 83
Figure 22: Domain Modelling 84
Figure 23: Analysis Model 100
Figure 24: Sequence Diagram for Analysis Model 101
Figure 25: Current Design Model (Rashev et al., 2000) 102
Figure 26: Proposed Design Model 103

1 Introduction

Industries often experience radical changes in seemingly brief periods of time. In most cases, these changes are imposed on them out of necessity. Today, the software development industry is facing a similar situation. Competitive demands are pushing software developers to create products, faster than ever, in a greater variety. In an effort to suggest appropriate solutions, a fair amount of research has been, and still is being carried out to identify the key ideas whose adoption can radically alter the way software developers do their jobs, with expected major gains in increased productivity and improved quality of products.

The concept of a software product line (SPL) is a promising approach to cope with what is often referred to as a software crisis. According to the Software Engineering Institute "A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way." (SEI, 2002). An SPL aims to achieve reuse of the following software development artefacts: (1) requirements documentation, (2) architectural design, (3) software components, (4) system modelling and analysis, (5) testing, and (6) planning. Thus, the creation of a new member of a product line involves little or no redesign and recoding and significantly reduced testing, making it possible to create high-quality software applications much more quickly than with a traditional development process.

The adoption of an SPL involves a conversion from a software development process that is characterized by developing an individual system and then creating variations of it, to a process that creates product lines and families of systems. One of the main difficulties is that doing so involves a change both in software development techniques and the organization of a company. This suggests that the transition to an SPL should be carefully explored and a tool that provides support throughout the whole process would be highly beneficial.

Another issue that has to be addressed is how to understand and track progress when we carry out a transformation to SPL development. A way to describe the artefacts in this process, the activities that are performed during the process, the operations that we use to manipulate the artefacts, and the roles played by personnel during the process should be provided. In this way, we can identify and describe the activities that may proceed concurrently and the activities that must be performed sequentially. Keeping all the information up-to-date in the face of frequent changes without automatic support will be a tedious and error-prone process.

There are a few existing methodologies for adopting an SPL approach. Even thought they differ in the extent to which some practices should be followed, the structure of the process remains the same. Roughly speaking, an SPL comprises two phases: domain engineering and application engineering. The first phase deals with defining the boundaries of the product line, analysing this domain for commonality and variability, and developing the core artefacts for this domain that will be reused in the production of the product family members. The second phase deals with eliciting the requirements for a specific product member, matching them with existing core assets, and finally producing the product member by exploiting the existing core assets as much as possible.

The focus of this thesis is to analyse, design, and implement a software product line support environment for the Sherlock methodology (Chapter 3). Sherlock is a domain-oriented methodology for an SPL that presents a more lightweight approach to SPL that can be introduced over an existing software development process. The support environment is designed to support the full software product line lifecycle as well as introduce novel features to address aspects missing in existing support tools (Chapter 4). The consolidation of the Sherlock methodology with a support tool is believed to alleviate the risks of software process overhaul and at the same time provide automated support for mining and developing reusable components within the domain of the product line.

The thesis is organised into 7 chapters. Chapter 2 provides an overview of three other SPL methodologies, which represent the most significant work in the field at the time of writing. Chapter 3 briefly describes the different phases of Sherlock. The core contribution of this thesis is offered in Chapters 4 to 6. Chapter 4 provides an analysis of the essential requirements for an SPL tool; Chapter 5 describes the design and implementation of the prototype tool, called Holmes. Chapter 7 presents a case study of domain analysis using Sherlock and Holmes. Finally, Chapter 8 draws some conclusions and identifies a few directions for future research.

2 Related Work

There are a few existing methodologies that address the problem of developing multiple products in a synergistic manner. Even though there are a lot of commonalities among them, these methodologies differ in how they identify the most critical aspects for the successful adoption of product line software development. Some approaches stress the organizational changes that have to be carried out during the transition to product line development, following Conway's Law that the design of a system represents the structure of the organization that has developed it. Other approaches put the emphasis on the changes in the software process that have to be followed in order to achieve a successful software product line without explicitly defining the structure of the organization that is to adopt the recommended process. Still others describe in detail the necessary augmentations in software development techniques that will effect the efficient production of product line members from the created product line environment.

2.1 RSEB

A holistic approach to adopting a Reuse-driven Software Engineering Business (RSEB) methodology is described by Jacobson (Jacobson et al., 1997). It aims at developing related software systems by exploiting high levels of reuse. In order to do so, a company should adopt a specific software development process, organizational structure, and software development techniques that allow for the creation and utilization of reusable artefacts from all phases in the software development process. RSEB also stresses the need for strong managerial involvement in the transition to reuse-driven software development in order to achieve and maintain the necessary changes in the company.

2.1.1 Obstacles in adopting large-scale reuse

The authors identify four major groups of obstacles in achieving large-scale reuse: engineering, process, organizational, and business-oriented issues.

Engineering – involve mainly technological deficiencies:

- The existing models lack clear identification of artefacts belonging to the requirements, analysis, design, testing, and implementation phases of the development cycle. Clear identification promotes reuse of these artefacts or, if needed, substituting them with reusable ones.
- Lack of reusable components resulting from poor techniques for packaging, documenting, and identifying components as well as insufficient mechanisms for searching components libraries.
- Lack of customizability in potentially reusable components. Usually, different applications have slightly different requirements for a given component. Consequently, if a component is rigid, it will seldom be chosen for reuse.
- Lack of tools to support the process of software reuse. Given the diversity of artefacts that have to be maintained and used, new and existing tools have to be integrated into reuse-oriented support environments.

Process – the traditional software development process does not provide opportunities for reuse. There is no explicit phase that forces an organization to analyse, design, and implement reusable components. Furthermore, the roles of the software architect and component engineer in the design and implementation of the basic technological infrastructure for reuse have not been identified.

Organizational – very few organizations systematically practice large-scale reuse. Large-scale reuse requires focusing on a few application systems covering one application area, i.e. a domain, at a time. In order to identify the components that will be shared among the applications in the domain, domain engineering has to be undertaken. This activity consumes additional time and resources. In contrast, many companies prefer to concentrate on one project in order to ensure development within the allotted time and budget. As a result of the latter approach, the possibility for reusing artefacts among projects is substantially reduced. Other factors that limit the adoption of large-scale reuse are a culture of not trusting others in an organization, the attitude of avoiding dependency, and the feeling of lost creativity.

Business – adopting and maintaining the practices of reuse necessitate additional investment. Performing domain engineering and developing a library of reusable components tie up capital until the projects that exploit the artefacts of these activities are completed. Additional funding is necessary to provide training and to obtain third-party tools necessary for the process of reuse.

Addressing and surmounting the abovementioned obstacles can only be achieved through the full support and involvement of senior management. Introducing partial solutions such as developing or buying a component library are not sufficient to overcome the inertia of developing software the old way. Jacobson et al. (1997) conclude that software reuse necessitates a major overhaul in the software process, organization, and development technique in place.

2.1.2 Changes in the process

Substantial reuse requires that the software process be based on two main groups of activities. The first group includes identifying reusable assets in terms of a system's architecture and the subsequent creation, appropriate packaging, and documentation to facilitate easy reuse. The second group includes the activities of implementing a software system as prescribed by the software architecture, using the components created. In order to do so, an organization needs to refashion its software process into four sub-processes that can partially be carried out in parallel. A brief description of each sub-process is provided in the following paragraphs.

Creating reusable artefacts. This process provides assets appropriate to the needs of the reusers. These assets may be new, reengineered, or purchased, including code, interfaces, architecture, and tools. This process may include activities such as domain analysis, architecture definition, technology evolution analysis, reusable assets testing and packaging.

Reusing the artefacts created in the previous process. The output of the reuse process is the creation of application systems as a coherent collection of reusable assets that are

structured according to the application architecture. This process includes the activities of constructing and testing complete applications.

Supporting the process of asset reuse. This process supports the overall set of processes by maintaining and facilitating the use of the reusable asset collection. The activities include certifying submitted reusable assets, classifying and organizing them into a library, providing additional documentation, and collecting feedback and defect reports from reusers.

Managing the process of software reuse. Includes planning, resource allocating, tracking, and coordinating the other processes. Activities entail setting priorities and schedules for new asset construction, and choosing an alternative way to proceed in the face of major changes.

Domain engineering is a key activity associated with the process of creating reusable artefacts. It provides a systematic way of identifying potentially reusable assets and an architecture to enable their reuse. The rationale behind domain engineering is that applications systems in one, and sometimes more, application domains share common functionality requirements. Consequently, these common requirements can be implemented as reusable assets that will be shared among the applications. The main difference between domain engineering and other system engineering practices such as Structured Analysis/Structured Design or Object-Oriented Analysis/ Object-Oriented Design is that domain engineering spans over several systems that belong to a single domain. There are a few domain engineering methodologies (Arango, 1994) that highlight different aspects of the process. They differ mainly in how they identify the scope of the domain and how well they match the target software process and technology.

Application system engineering is concerned with the process of building applications and consequently has existed since the beginning of software development. Originally, application systems were built from scratch or by using function libraries. Nowadays, the objective is to build software systems much more quickly and more cost-effectively than

before by stressing the extensive reuse of a set of reusable assets that have been provided. Thus, the activities involved in modern application engineering are customizing and assembling the provided assets into applications.

2.1.3 Changes in the organization

In a traditional software organization, senior management allocates resources over a number of projects. Each project manager runs his or her project, and there is no organized way of creating reusable artefacts. Systematic reuse necessitates the existence of a separate unit that is concerned only with the development of reusable artefacts. In many companies with experience in systematic reuse, a third function evolves – that of support. Thus, a company maintains three independent organizations that have separate managers (Figure 1).

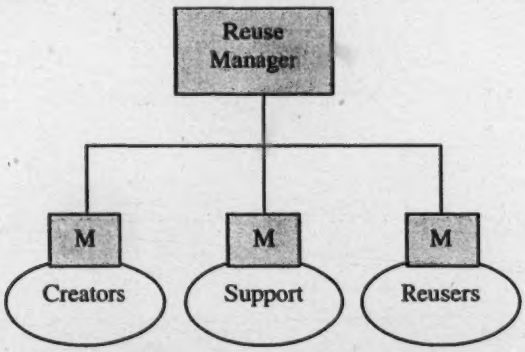


Figure 1: Organization Structure (Jacobson et al., 1997)

All the necessary changes that have to be implemented in the process and the organization of a company require much time, investment, and effort. So a company faces two conflicting pressures: on the one hand, it has to keep its current projects going in order to meet its financial objectives, and on the other, it has to reorganize itself and adopt a reuse-driven approach to developing software if it wants to remain competitive. An adoption strategy that has proven its viability in practice is implementing reuse strategies in pilot projects. If they prove successful, the reuse practices are gradually extended to other projects. A pattern often found in the industry is of incrementally adopting reuse practices (Figure 2).

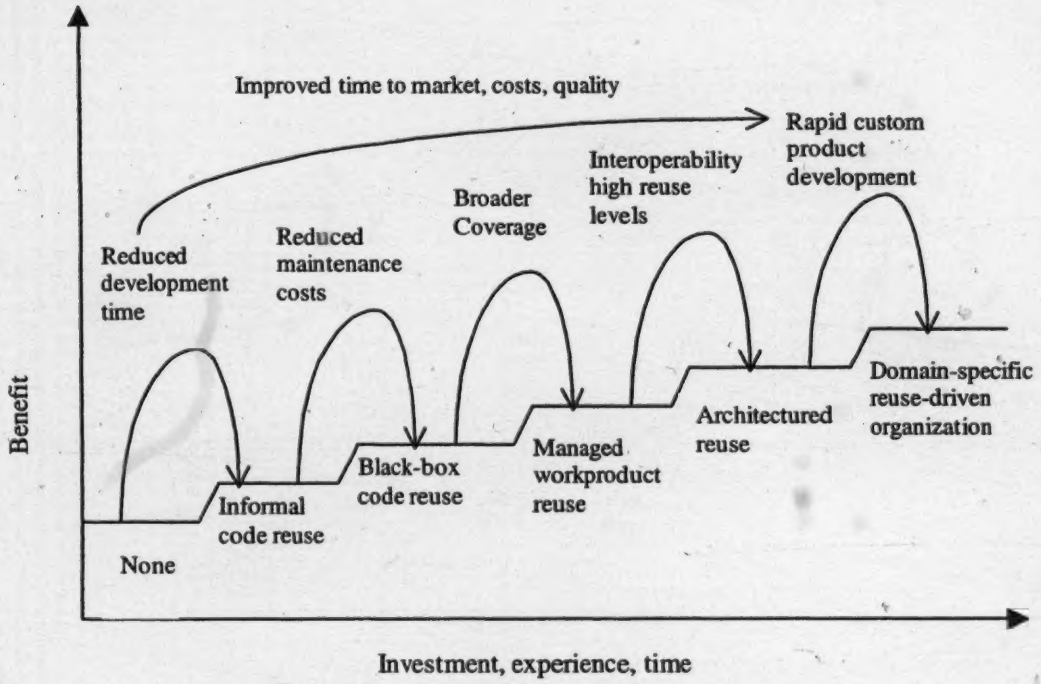


Figure 2: Incremental Adoption of Reuse (Jacobson et al., 1997)

This diagram depicts the general rule that an increased level of reuse shortens the time to market, lowers the cost of products, and increases the quality of the products. The transition of no reuse to informal reuse occurs when developers trust each other and a need for a shorter time to market is present. In this case, reuse is mostly based on copying code in different applications. This type of code reuse is very hard to maintain because any modification or defect fix has to be done multiple times and in most cases manually. Consequently, as the level of reuse rises, maintenance becomes more difficult.

The next level is "black-box" code reuse. In this case, the code is reused as a carefully packaged and documented module or component. Thus, practically all reusers use the same code and preserving the consistency of the code is not an issue. Since different applications can have slightly different requirements for a component, variations of the component should be supported. In this type of reuse, it is not clear who is responsible for the configuration management – should this be the person(s) who originally created the

component, or the one(s) who required the modification. The difficulties encountered in configuration management necessitate a transition to the level of managed workproduct reuse. At this level, a distinct organization is responsible for the creation and maintenance of reusable assets.

At the level of architected reuse, care is taken in advance to identify components that have to be used and to ensure that they will fit the architecture. Domain-specific reuse focuses on identifying the common components among the application systems in the domain.

In RSEB, the following principles are identified as essential in order to maintain large-scale reuse:

- Maintain top-management leadership and financial backing over the long term.
- Plan and adapt the system architecture, the development processes, and the organization to the necessities of reuse in a systematic but incremental fashion. Start with small pilot projects, and then scale up.
- Plan for reuse beginning with the architecture and incrementally developing the reusable components.
- Move to an explicitly managed reuse organization that separates the creation of reusable components from their reuse in application systems and provides an explicit support function.
- Create and evolve reusable components in a real working environment.
- Manage application systems and reusable components as a product portfolio of financial value, focusing reuse on common components in high-payoff application and subsystem domains.
- Realize that object or component technology alone is not sufficient.
- Directly address organization culture and change using *champions* and *change agents*.
- Invest in and continuously improve infrastructure, reuse education, and skills.
- Measure reuse progress with metrics and optimize the reuse program.

2.1.4 Object-Oriented System Modelling

The RSEB methodology heavily relies on the Unified Modelling Language (UML) (Booch et al., 1998) through all phases of system development: requirements capture, robustness analysis, design, implementation, and testing. During each activity, the understanding of the system is refined and its functions are described in more detail. For each phase, several models are used.

The phase of requirements capture is based on the use case model, which consists of an actor and a use case. The actor is any entity that interacts with a system by exchanging data and events. The use case is a sequence of transactions performed by a system, which yields an observable result that is of value for a particular actor. The use cases A and B can be connected with two types of generalization relationships: <<uses>> and <<extends>>.

- <<uses>> - indicates that B inherits A, and consequently an instance of B can perform all the behaviours of A.
- <<extends>> - indicates that an instance obeying use case A may at some time discontinue obeying A and instead start obeying use case B temporarily. After the instance of A has finished obeying B, it will resume obeying A.

Connecting use cases by relationships helps the software engineer to identify common functionalities in the customer's requirements very early in the development process and thus to identify the potentially reusable components.

During the phase of robustness analysis, an analysis model is created. It includes the following stereotypes:

- <<boundary>> – defines the interface between a system and an actor
- <<entity>> – is a long lived object in the system
- <<control>> – performs use-case specific behaviour

These types are used in constructing collaboration diagrams, which show how the analysis objects interact in performing a use case. A collaboration diagram is associated with one or more use cases, and this connection is made explicit by a <<trace>>

relationship. Another type of diagram, a sequence diagram, shows in more detail the type of messages that will be exchanged among the types while performing a use case. Similar to collaboration diagrams, sequence diagrams are associated with one or more use cases.

In the next step, a design model of the software system is created. It consists of design classes, which contain more details related to the target language and execution environment. The design model is derived from the analysis model by mapping each analysis type to one or more design classes. In the case of multiple classes, the relationship among them is defined. Depending on the target language, it can be inheritance, aggregation, association, delegation, etc. Also, a trace relationship between the analysis type and the design classes is established.

Lastly, the implementation model is represented by the source code. Each of the classes or modules in the implementation model closely matches the one in the design model, but in addition each has method/function bodies.

The test model is represented by test cases. Each test case can be regarded as having an instantiation with an expected result. Use cases represent excellent instruments for test planning by providing the basic cases that need to be tested. Thus, the system model represented by the UML notation provides traceability from the requirements specification represented by the use case model to the phase of testing represented by the test model.

2.1.5 Application Family Development

Jacobson (Jacobson et al., 1997) identifies three types of application system families:

- An application system suite – a set of application systems that are intended to work together to help some actors accomplish their work, e.g. Microsoft Office
- Application system variants – the same application system needs to be configured, packaged, and installed differently for different users, e.g. telecommunication switching systems

- A set of fairly independent application systems – can be treated as a family when built from the same set of low-level reusable components, e.g. applications built using the Microsoft Foundation Classes (MFC) Framework.

In RSEB, there are three main activities that constitute the development process: application family engineering (AFE), application system engineering (ASE), and component system engineering (CSE). The process of AFE gathers requirements from a few customers and transforms these requirements into a suite of application systems. The workers involved need to understand what current and potential customers will need and want in the future in order to envision appropriate use cases. The use cases are then transformed into a suitable architecture through robustness analysis and design. This architecture should define the candidate components and application systems that will be needed. Alignment with existing products, such as legacy systems, GUI frameworks, or middleware object request brokers, is performed to explore the possibilities of reusing existing software. During AFE, decisions are taken as to when to initialise CSE and ASE.

CSE designs, constructs, and packages components into component systems. It begins with requirements capture, which is based on information from a wide range of sources such as business models, domain experts, and application end users. After the requirements are analysed for consistency, commonality, and variability, the results are used to incrementally design, develop, and test the components. CSE concludes with certifying and packaging the components for reuse and easy retrieval. The reusers of these components should follow a process and use tools provided by component developers.

During ASE, applications are built by selecting, customizing, and assembling components from the component's systems. This process also includes requirements refinement for a specific application and proceeds with incremental analysis, design, implementation and testing. It is realistic to expect that even if application engineers try to reuse as many components as possible, there will be features that necessitate the

development of new code. In their work, application engineers use the tools, modelling languages, and process instructions provided with the components system.

2.2 PuLSE

Product Line Software Engineering (PuLSE) is an approach that builds on existing methods of domain engineering (SPC, 1993) (Ardis and Weiss, 1997) (Kung et al., 1990) to define a methodology for software product lines (Bayer et al., 1999). The authors of this methodology claim that domain engineering approaches have had mixed success when applied in industry. Consequently, a new methodology that overcomes their shortcomings and facilitates product line development is needed.

2.2.1 Shortcomings of domain engineering methodologies

The rationale behind domain engineering is that by focusing on a specific domain, where applications significantly overlap, a large-scale reuse can be achieved. According to the authors of PuLSE, domain engineering methods have not proved as effective as expected. The reasons fall into three main groups:

- Misplaced focus on domains as opposed to products
- Deployment complexity
- Lack of customizability

Domain engineering relies on the notion of an application domain to determine the boundaries of the reusable infrastructure. An application domain includes all possible applications that can be developed in that domain. Domains have proved difficult to scope and engineer since they include many extraneous elements that have no value to an enterprise. Consequently, the domain view provides little economic basis for making the right scoping decisions. Instead an enterprise should focus on a group of particular products, including existing, under development or anticipated products.

The deployment complexity derives from the overstated focus on organization issues and the lack of technological support. Often, domain engineering methods assume that the technological problems of how to scope, model, and architect the reusable infrastructure

have existing solutions. The authors of PuLSE believe that in practice this is rarely the case, and consequently the focus of a product-line methodology should be shifted.

The authors claim that since different companies have different needs, applying the same methodology in different industrial settings can be inefficient. The existing methodologies are either not flexible enough or are too general. Thus, they cannot be applied successfully in practice without additional modification and expert support.

2.2.2 Overview

PuLSE consists of three groups of elements as shown in Figure 3: deployment phases, technical components, and support components.

Deployment phases include the stages of product line evolution. They describe the activities included in the instantiation and usage of a product line. The phases are:

- Initialisation: gather information about the company to adopt product line development and customizes PuLSE according to the company's specifics
- Infrastructure Construction: define the boundaries, model, and architect the product line infrastructure
- Infrastructure Usage: use the reusable assets comprising the product line infrastructure in creating product line members.
- Evolution and Management: evolve and manage the infrastructure over time

Technical components provide the technical support needed to carry out the product line development. They are used throughout the deployment phases. There is no strict correspondence between each deployment phase and each technical component. Nevertheless, some technical components are used only in one deployment phase. The technical components are grouped into following sections:

- Customizing: used during the Initialisation phase
- Scoping: used in defining the boundaries of the product line infrastructure by focussing on product definitions

- Modelling: used in representing the product characteristics in the product line by a single notation and denoting the separate product family members
- Architecting: used to develop the reference architecture of the product line and to maintain traceability with the product line infrastructure model
- Instantiating: used in performing the usage phase
- Evolving and managing: used in configuration management, adapting assets that do not fit the product line, and evolving the existing product line infrastructure

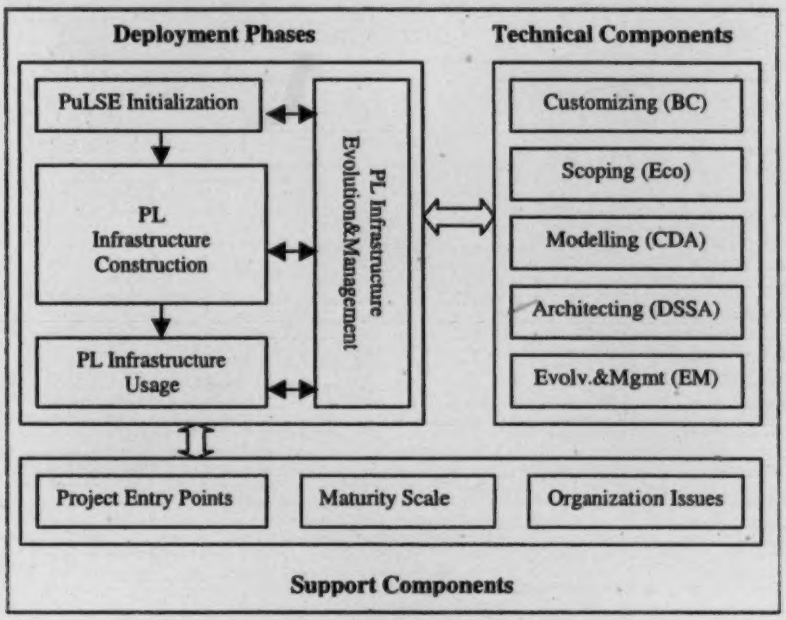


Figure 3: PuLSE Structure (Bayer et al., 1999)

The support components are documentation and guidelines compiled and provided to the user in order to enable a better adoption, evolution, and customization of the product line. These components are used by the other elements and consist of:

- Project Entry Points: information representing the different project types. For example, are there already existing assets that need to be integrated in the product line or are there multiple projects that have been developed independently but possess significant overlap that makes adopting the product line approach beneficial?

- Maturity Scale: guidelines for adopting and evolving a product line approach in a company by using PuLSE.
- Organization Issues: guidelines for creating and maintaining the right organizational structure for developing and managing product lines.

2.2.3 Infrastructure Construction

The initialisation phase produces a tailored version of PuLSE to suit the specifics of the company adopting it. This phase itself consists of three sub-phases: baselining, evaluating, and customizing.

During *baselining* information necessary for customizing PuLSE is gathered. This information is gathered according to *characterization factors*, which define the specifics of the organization adopting the product line approach, such as the type of projects it is developing, the amount of resources it has, etc. The relevant customization factors are chosen by following the guidelines included in the support components. Each characterization factor possesses a baselining strategy that governs the acquisition of necessary data. The information gathered for all the characterization factors constitutes the current profile of the company. This profile is analysed for dependencies among the characterization factors during the evaluation. In this sub-phase, the effect of these factors on the components of PuLSE is determined. This results in the raw instantiation profile for the customizations that need to be carried out. In the next step, the sub-phase of customization targets at deriving a complete product-line process. At this stage, decisions about the expected number of iterations as well as the final process definition is passed on to the technical component for evolving and management (PuLSE-EM).

The scoping component (PuLSE-Eco) is used to identify the scope of the product line. As a first step, the possible products to be included in the product line are identified. These products are used to identify the characteristics and validate them. The information about the characteristics and the products is organized into product maps. The main idea of PuLSE-Eco is to use the business objectives of the stakeholders of the product-line to define which products will be finally included in the product line. Since these objectives

are in most cases too abstract to determine within PuLSE, they are augmented into evaluation functions, which include characterization and benefit functions. Characterization functions evaluate the characteristics of each product, whereas the benefit functions, using the characterization functions, determine the most beneficial characteristics and products that the product line will cover. During the step of characterizing products, the characterization functions are applied to the product/characteristic combinations. Finally, this information is added to the product maps.

The next step of benefit analysis is the core step of PuLSE-Eco. At this, point the information gathered is used to identify the scope. The values of the characterization functions are used to assign values to the benefit functions. In order to produce a single scope definition, the values of the different benefit functions need to be balanced. Doing so represents a classical multi-objective decision problem. The PuLSE-Eco doesn't provide a specific technique to address the problem but suggests using existing techniques (Mollaghasemi and Pet-Edwards, 1997). The final workproduct of PuLSE-Eco is a product plan, which is fed to the PuLSE-EM component.

The technical component for modelling (PuLSE-CDA) is aimed at creating a product line model. To achieve this an additional refinement of the economic scope of the product line is performed, if necessary. The specifics of this process are determined during the phase of initialisation (PuLSE-BC). The core activities in PuLSE-CDA are eliciting, structuring, and modelling the relationships among the elements of the product line infrastructure. Two types of workproducts are defined in the phase of modelling: storyboards and other domain workproducts. Storyboards are used to represent the flow and type of actions constituting an activity of interest in the domain. Examples are workflow diagrams and message sequence charts. Other domain workproducts may include models to capture additional characteristics of the product line, such as data models to represent common or varying data structures to all product members.

The product line model is created by first eliciting and creating storyboards for the various product family members. Next, the commonality among these storyboards is extracted in the form of generic storyboards that apply to all product members. The variability is also explicitly represented in other workproducts. Each product member is then derived from the generic storyboards by applying a decision model. In this decision model, variability is represented as a decision that has to be taken among few possibilities. Thus, the specification of a product member is derived from the generic one by resolving all decisions (variabilities) in it.

The technical component for architecting (PuLSE-DSSA) focuses on creating a domain-specific software architecture that can be used to instantiate current and future product family members. The basic idea behind PuLSE-DSSA is the incremental development of the reference architecture guided by scenarios. The scenarios are either generic, which represent functional requirements, or property-related, which represent non-functional requirements. The generic scenarios are derived from the generic storyboards and other workproducts created in PuLSE-CDA. Each generic scenario is further augmented by a number of property-related scenarios that are associated with it and is ranked according to its architectural importance.

The iterative development of the architecture starts with a subset of the identified scenarios that have the highest architectural importance. The current iteration is completed when all scenarios from the current subset are fitted into the architecture. The inclusion of a generic scenario may result in more than one architectural solution. In such a case, the property-related scenarios are used to further evaluate each candidate. Optionally, a prototype of a candidate can be built to evaluate more precisely its viability.

During the creation of the reference architecture, each decision to be taken has to be further refined during the stage of implementation, and the anticipated possible solutions have to be recorded. The decisions to be taken and their possible solutions are recorded in a configuration model for the reference architecture.

2.2.4 Usage

The Usage Phase of PuLSE aims at specifying, instantiating, and validating a product family member. As a first step, the customer's requirements for the specific product member are gathered, analysed, and mapped to the existing characteristics and products in the product map as created in the PuLSE-Eco. The level of overlap between the planned product member and the product map, together with existing historical data concerning product line infrastructure usage, helps to determine what extent of the functionality of the new product member will be covered by already developed assets. This information is used in determining the necessary resources and the schedule for the instantiation of this family member.

In the next step, the product line model or a part of it that corresponds to the application under development is instantiated by using the decision model created in PuLSE-Eco. Thus, the product member specification is derived and validated against the customer's requirements. The validated specification is added to the configuration history of the product.

The next step is instantiating and validating the family member's architecture. Similar to the previous step, the architecture for the family member is derived from the product-line reference architecture using its configuration model. Then the architecture is validated against the product's specification, and it is added to the product's configuration history.

The actual implementation of the family member may include components that come from three sources: reusing an existing component from the product-line infrastructure, developing a new component that will be added to the product-line infrastructure, or developing a new component that will be not be added to the collection of reusable assets. The decision between the last two options is taken in the PuLSE-EM component. Again, the design and code of the family member are validated against the architecture and are entered into the product's configuration history.

Finally, before deployment, the product is validated against the acceptance tests that are performed by the customer. Failure at this level triggers a new iteration of the Usage Phase.

2.2.5 Evolution

The purpose of the evolution phase is to monitor and control the evolution of the product line infrastructure over time. The evolution phase uses the technical component PuLSE-EM, which is customized to the requirements of the specific company. PuLSE-EM is also used in the phases of initialisation, construction, and usage. PuLSE-EM is used not only in maintaining the workproducts from the different phases but also in coordinating the activities from these phases and consolidating their assets.

Based on the information gathered during the phase of initialisation, the evolution phase plans and guides the development of the product line infrastructure. In any of the sub-phases of product line construction and usage, a modification to existing assets that belong not only to the preceding but also to the following phases may be necessary. The role of the evolution phase and its supporting technical components is to evaluate the benefit of this change request, determine its impact on other existing assets, and finally propagate the necessary changes by instantiating the associated parts of the involved technical components.

2.2.6 Support

Supporting PuLSE provides the guidelines to customize the methodology and help the company in adopting a product line approach in both development and organizational aspects. To do so, the guidelines are organized into three groups: Project Entry Points, Maturity Scale, and Organization Issues.

The project entry points describe the usual contexts in which PuLSE is adopted:

- Pure PuLSE: it is characterized by a situation in which a new product line is set up in a company. In this context, all components can be established with full traceability among them
- Evolutionary PuLSE: the different components of the methodology are adopted incrementally in an ongoing development process
- Reengineering-driven PuLSE: existing software systems that have been developed individually are grouped into a product line. In this case, existing assets are used to instantiate the PL infrastructure construction phase

The PuLSE maturity scale is designed to help an enterprise adopt the methodology and measure its maturity within the product line context. The PuLSE maturity scale is based on the maturity scale defined in the Reuse Adoption Guidebook (SPCSC, 1993) and identifies four maturity levels:

- Initial: only single PuLSE technology components can be applied at a time – mainly Eco, CDA, DSSA after the necessary customization
- Full: All technological components involved in the Construction Phase are used; nevertheless, their degree of integration can vary
- Controlled: PuLSE is applied as a complete development lifecycle. Traceability among the different phases is established and maintained.
- Optimizing: PuLSE methodology is further refined over a number of iterations after its initial adaptation

The organizational issue guidelines cover both development and project organization areas. For the development organization, the recommendations are:

- Divide the application development into areas of specialization according to the separation of concerns
- Vertically layer the application development into analysis, architecture, implementation and deployment
- Assign permanent personnel to the areas of specialization within the vertical layering whenever possible
- Supervise and enforce the process rules, architecture soundness and evolution

For the project organization, the guidelines are:

- Dynamically assign personnel to projects according to their responsibility area, i.e. developers belong to an area of expertise, not to a project
- Keep project leadership very small. Project leaders coordinate the project development within the product line
- Allocate 70% of developer's time to current project development and 30% of their time to the evolution of the reusable infrastructure

2.3 FAST

The authors of the Family-Oriented Abstraction, Specification, and Translation (FAST) process (Weiss and Lai, 1999) propose a systematic approach to analysing potential families of software systems and developing facilities and processes for generating members of the product family. The FAST process is built on two main sub-processes (Figure 4): domain engineering, which creates the facilities and defines the processes used for fast generation of product members; and application engineering, which produces the members of the product family by using the facilities created in domain engineering.

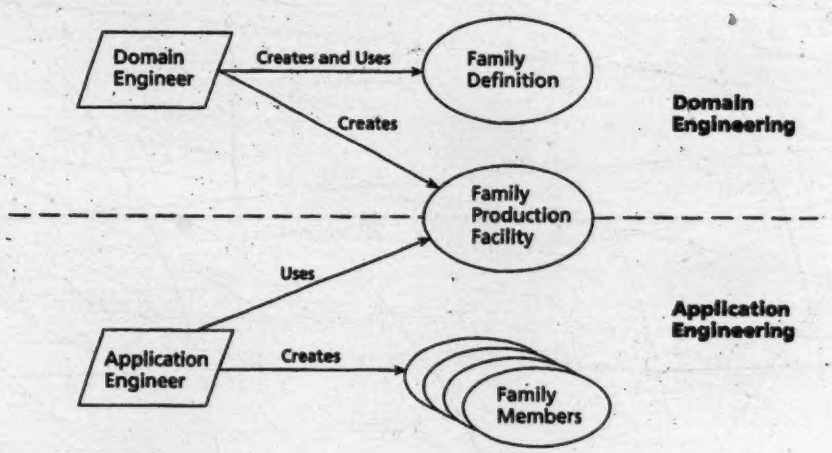


Figure 4: FAST sub-processes (Weiss and Lai, 1999)

2.3.1 Careful Engineering vs. Rapid Production

Software engineers face a continuing dilemma. On one hand, they are asked to develop software systems that attract customers with their rich functionality, ease of use, and reliability, while at the same time being easy to maintain and extend, according to future customers' requests. Such products can be achieved only after careful engineering. On the other hand, software engineers are pressed to develop software products in the fastest possible way, so that the products can be marketed ahead of competition. In most cases rapid development, as it is practiced nowadays, is contrary to careful engineering, and what is sacrificed is the activities and artefacts that enhance a product and its constituting parts in terms of their later lifecycle maintenance and extension.

Today, few companies succeed in striking a balance between schedule pressure and careful engineering. To achieve this balance, they should adopt a systematic approach to developing software products. Engineers in other fields have experienced the same conflicting goals of rapidly producing carefully engineered products. Many of the solutions that have been adopted are based on the idea of developing a family of products that can be built by exploiting the same production facility. In this case, a family denotes "a set of items that have common aspects and predicted variabilities" (Weiss and Lai, 1999).

In order to create a profitable product family that results in decreased development time, a few issues need to be addressed. FAST was designed to address the following problems (Weiss and Lai, 1999):

- Ill-defined and changeable requirements either due to customer's uncertainty or missing information
- Confusion of requirements, design, and code
- The need for rediscovery and reinvention
- Adapting existing legacy systems to new technologies and requirements
- Redundant specifications

2.3.2 Principles of Family-based development

FAST is based on three main assumptions concerning the development of software systems (Weiss and Lai, 1999):

- The redevelopment assumption. Most software development re-implements existing software systems or, more precisely, builds variations of and/or extensions to these systems.
- The oracle assumption. It is possible to predict to a significant extent the type of changes that will be required to a software application in the future. In particular, it is possible to predict where the variations will occur.
- The organizational assumption. It is possible to organize both the software and the organization that develops and maintains it so as to limit the necessary changes of any predicted modification to a small number of software modules and involve a small number of developers. Thus, the task of producing new versions of a system can be carried out as relatively independent modifications to the system performed by different developers, i.e., the various modifications can be performed concurrently.

From the redevelopment assumption it follows that by avoiding the re-implementation of common modules, we can decrease our total development time. Identifying the commonality among the software systems is the key in defining a product family. The oracle assumption states that the variability in the software systems can be predicted, and consequently, we have a good understanding of what type of systems will be built. The organization assumption states that we can structure our systems and organization in a way that will permit efficient implementation of predicted modifications.

The abovementioned high-level characteristics of family-based development can be realized through already existing methods and technologies. Examples of these are abstractions, information hiding (Parnas, 1972), and predicting change based on past experience. Choosing good abstractions is essential for ensuring extensibility in the future. Abstractions represent common entities by providing an interface to manipulate them and hiding the variability among them behind this interface. In FAST, abstractions

are used both in the design of the family architecture and in constructing the application modelling language. These abstractions become the basis for building adaptable components that are used in generating the family members. The information hiding principle includes the subset of abstractions that hide the decisions most likely to change. Thus, the choosing the modules to implement information hiding and predicting future changes are the main decisions on which the product family architecture is based.

Successful prediction of future change can be based on two sources: past experience in maintaining a software system and prediction of marketing or technological trends. By examining the modification history of a software system, a software practitioner can identify the modules of the system that have undergone the most change. Weiss et al. (1999) stated that these modules should certainly be domain engineered because they represent both a bottleneck in the development and provide revenue. Such modules can be considered bottlenecks since they have utilized a lot of development effort and possibly have slowed down the implementation of other modules dependent on them. Additionally, they are vital for the company because in many cases, the modifications are in response to requests for new features. Therefore, such modules should make heavy use of abstractions and information hiding in order to limit the dependencies of other parts of the system on them.

Another source that can be used in predicting future changes is the people involved in monitoring market trends. They can often provide valuable information about the possible evolution of user expectations in the future and thus provide direction concerning new features that may be required for implementation. The information from early adopters of new technologies can also be used to evaluate the possibility of adopting a new technology that can streamline the development and make possible the implementation of new services.

The organizational assumption follows Conway's law that the structure of a software system mirrors the structure of the organization that has developed it. Consequently, in order to limit the dependencies among software modules, the company has to be divided

in separate organizations that develop different modules of the system. Each organization uses services provided by a module developed in a different organization through a standard interface. The products within a product family often span multiple domains. Accordingly, each organization within a company may be responsible for developing the software assets belonging to a single domain. In this case, each family member will be developed by integrating components coming from different domains. Often, doing so necessitates the creation of an additional organization that deals only with producing the family members. The adoption of such an organization may result in additional gains when a company develops multiple product families and some of the domains are shared among the different product families.

2.3.3 Overview of FAST

FAST includes three main subprocesses:

- Identifying families worth of investment (Qualify Domain)
- Investing in facilities for rapid production of family members (Domain Engineering)
- Using those facilities to produce product line members (Application Engineering)

The first step is qualifying the domain, a process which includes an economic analysis and estimating the number of family members; the objective is to ascertain whether it will be beneficial to adopt a product line approach for a specific situation. The basic economic assumption in FAST is that investments in engineering the family will be paid back by the more efficient production of family members. This results in discriminating between two cases: one in which little or no attention is paid to domain engineering, and a second one in which the domain is engineered with the intent of making the production of family members more efficient. In the former case, the cost associated with the production of a new family member stays almost the same. In the latter case, a new family member is produced at almost no cost by utilizing the facilities created during domain engineering.

In addition, when product families are developed by performing domain engineering beforehand, a company faces making an initial, substantial investment to create the product family environment. During the subprocess of domain engineering no income can be expected from the activities carried out and the artefacts created. This means that a company has to perform a careful economic analysis and decide whether it is profitable to continue with product family development. The risk associated with product families can be represented by the following economic model. (Figure 5). Assuming that the cost of producing one family member is M, then producing N members will cost C = MxN. This cost is relatively evenly distributed in time. Conversely, if domain engineering is performed, a company has to start with the initial investment – I. Assuming that producing a family member after domain engineering has been performed costs FM, then the cost incurred to produce N family members is C = I + FMxN. In order for the product family to be viable, FM has to be substantially lower than M. The difference depends on how much automation is achieved by the artefacts produced during the phase

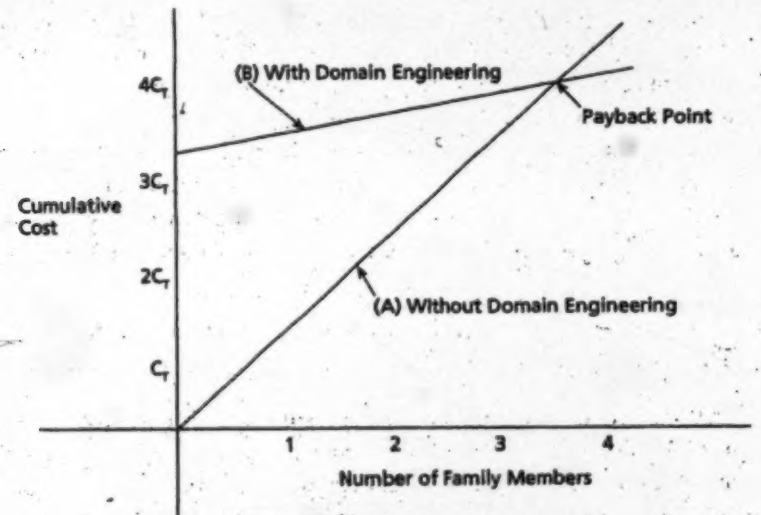


Figure 5: Economic Model (Weiss and Lai, 1999)

of domain engineering. It has to be pointed out that there are certain activities that will never be automated. For example, the application engineer will always interact with a customer to determine the requirements for a new family member and must be able to demonstrate to the customer that those requirements have been met.

The slope of the line (B) depends on the amount of investment done during the phase of Domain Engineering and on the specifics of the domain itself. Some domains lend themselves very well to automation; others include more activities that have to be manually performed. In order to alleviate the risk of making a substantial initial investment in Domain Engineering that later cannot be paid back, FAST recommends incremental adoption of Domain Engineering and iterative creation of the application production environment. As the company's experience with Application Engineering grows and proves beneficial, it can further invest and improve in its application environment facilities.

FAST does not prescribe a specific Domain Engineering methodology but leaves open the possibility for choosing one depending on what is considered most appropriate in a specific situation. Still, certain activities and artefacts are essential in FAST and have to be included in each FAST instantiation. The activities that have to be included are the following:

- Defining the family (domain)
- Developing a language for specifying family members – application modelling language (AML)
- Developing an environment for generating family members from their specifications
- Defining a process for producing family members using the environment

The artefacts of Domain engineering should include:

- An economic model of the domain
- A definition of the family, including a dictionary of the standard terminology used, the commonality among the members of the family, and the variability in the family
- A description of the decisions that have to be taken in order to specify a product member, i.e., choosing a specific decision for each point of variability that the product member has.
- A specification of AML: composition- or compiler- based

- Tools necessary in the process of application engineering, including analysis, modelling, documentation, and source code generation tools.
- A definition of the process to be used in application engineering

The phase of Application Engineering focuses on grasping the customer's requirements and using the application environment to model and generate the desired software systems. The application environment created in DE should enable the application engineer to abstract from design and coding decisions and concentrate on translating the customer's requirements into an application model. The application model is defined in AML, which is part of the application environment.

The application engineering phase is an iterative process. After the initial model of the application system in created, this model is analysed by using the tools included in the application environment to validate that it satisfies the customer's requirements. The source code for the application system is generated from the validated model and the generated implementation is given to the customer for acceptance testing. The same steps are repeated until the customer is satisfied. In the same process, the application engineers give feedback and suggestions for improving the application environment.

2.3.4 Process model of FAST

The FAST process, as with any other engineering process, is a sequence of decision-making activities. An engineer needs to know what decisions to make, when to make them, what the results mean, and how to present them. By providing a good description of a process, we can be confident that all essential steps in a process will be covered. Additionally, it makes it possible to repeat a process in different environments and projects. A systematic description also gives us the possibility to record any modifications that have been introduced to the process in order to improve it or customize it for a specific environment.

In the case of product family software development, characterized by the myriad of activities it comprises, the need for providing a detailed model of the process is even more acute. Such a model will also facilitate the creation and adoption of automated support which can substantially help in sharing and managing the artefacts of the process. Consequently, the goals of the FAST model, called Process and Artefact State Transition Abstraction (PASTA), should be accurate in describing the work products and the task of the engineers; provide a criteria for assessing when a work product is completed; and on the other hand provide for sufficient customizability. A very strict and detailed process is hard to adopt in different companies. In order to address this issue, PASTA does not specify a method for carrying out domain engineering and provides only partial ordering of the activities that are defined.

The main abstraction used in PASTA is state machines. All decision-making activities and artefacts are represented as state machines that can sometimes be executed in parallel. In order to provide an appropriate level of detail and avoid making the model incomprehensible, the state machines are organized hierarchically. At higher levels, each activity is represented as a state in the state machine on that level. When necessary, each activity can be explored in detail by opening the associated state diagram that models the different stages in that activity. Thus, two different types of states representing processes (P-states) are defined in PASTA: elementary and composite. The FAST process is defined in three hierarchical levels: the bottom level consists of elementary states representing activities that are performed on artefacts; the middle level consists of composite states; and the root level is the aggregate FAST process (Figure 6).

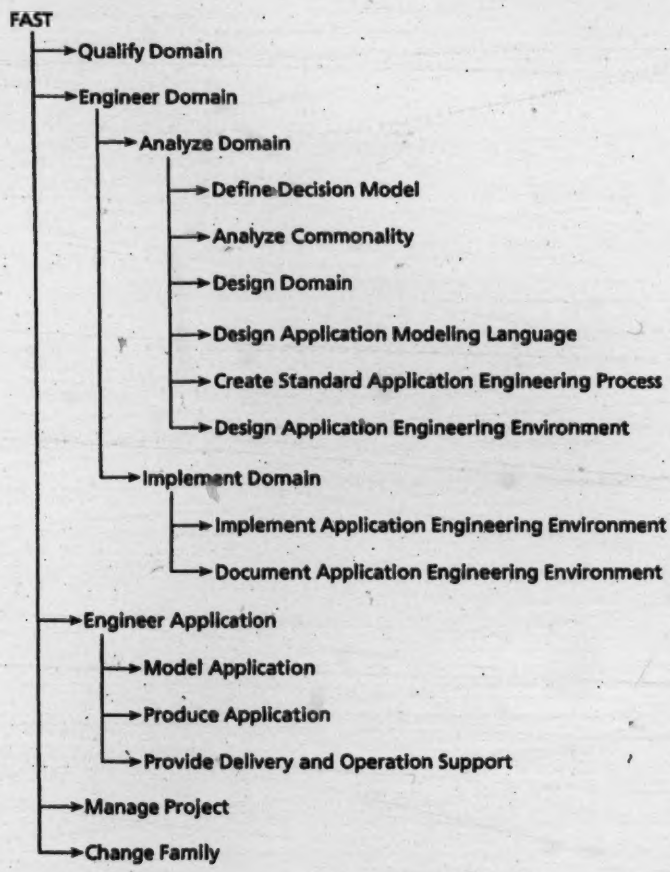


Figure 6: The FAST hierarchy

The artefacts are represented by state machines in which each state represents a milestone in the development of an artefact. Examples of these states are:

- Referenced – the artefact is referenced somewhere else in the process and consequently, if non-existent, it has to be created
- Defined and Specified – the artefact has been developed
- Reviewed – the artefact has been reviewed and accepted

Similarly to processes, artefacts can be composite or elementary: the composite artefacts consist of multiple elementary artefacts. Contrary to the process hierarchy, the artefact hierarchy does not have a limitation on its depth.

2.4 Summary

The three methodologies described in this chapter represent different approaches that can be undertaken towards adopting software product line development. RSEB defines a very thorough approach to ensuring high levels of software reuse. Large-scale reuse is achieved by imposing specific practices and rules for the software process, the company's structure and the software development techniques. The software process includes specific activities for creating, supporting, and using reusable artefacts. In addition, the activities of domain engineering and application system engineering are defined. The former ensures that proper reusable artefacts are created, and the latter facilitates their use during the phase of application construction. The structure of a company mimics these activities by providing separate organizations for developing, supporting and using the reusable artefacts. The software development techniques including requirements capture, business modelling, and software architecture are based on the Unified Modelling Language (Booch et al., 1998). As a result, a company that wishes to adopt software product line development following RSEB must carry out an overhaul of their business.

The PuLSE methodology has evolved as a methodology of its own through the usage of other existing practices for domain engineering. Its main difference from the other domain engineering methodologies is that it places the focus on the products that a company wants to develop rather than on a domain. In this sense, the product line scoping is done via *benefit analysis*, which assigns each product a benefit value, and based on such a *product map,* the products to be included in the product line are defined. Another characteristic feature of PuLSE is its separation into Deployment Phases, Technical Components and Support Components (which are described in more detail in section 2.2.2). The technical components provide technical support for the different practices employed in the software product line development. A difference between PuLSE and RSEB is that in PuLSE, there is no strict mapping between the company's structure and the software product line practices. For example, as the authors of PuLSE mention, the creation and maintenance of reusable artefacts can be handled by the same team by devoting 70% of their time to development and 30% to maintenance, as opposed to the clear separation of such teams in RSEB. In short, PuLSE is concentrated on the changes

needed in the software process rather than on the structure of the organization and its development techniques. In order to provide a flexible framework for modifying the software process an explicit phase of customization is performed, a step that is non-existent in RSEB and FAST.

The FAST methodology is targeted at developing a family of systems consisting of variations of the same system. In this case, the main effort is expended in searching for the common aspects among these variations and trying to predict the possible variabilities that can occur in the future. A pivotal point in FAST is the choice and development of an Application Modelling Language, which permits fast generation of product family members by exploiting the artefacts created during the phase of domain engineering. A characteristic feature of FAST that sets it apart is the simple but explicit economic model that defines the economic benefit of starting a product line development. The software process in FAST is divided into two main phases: domain engineering and application engineering. Similar to PuLSE, no explicit guidelines of how these sub-processes should be mapped to an organization's structure is provided. Even though FAST does not prescribe a specific Domain Engineering (DE) methodology, it specifies the type of artefacts that should be present after DE is performed. Overall, FAST provides a systematic definition of the software process and artefacts but gives few details about the organization's structure. The main difference between FAST and the other two methodologies is its emphasis on software techniques, which should be based on product member generation rather than creation.

Even though the three presented methodologies possess unique features, they all can be described as "heavyweight", i.e., they necessitate major changes in the way a software company works, require substantial initial investments, and are more or less applicable only to medium and large companies. In the next chapter, a more "lightweight" methodology that facilitates the product line development is presented. It can be adopted as an augmentation/extension of an existing software process and techniques, thus substantially alleviating the risk of major process modifications and organization investments.

3 Sherlock – a methodology for adopting SPL development.

Sherlock is a fully developed object-oriented domain analysis and engineering (DAE) methodology targeted at developing a set of products within an analysed domain (Predonzani, et al., 2000). It spans over all activities involved in DAE – from market analysis to system implementation. A characteristic feature of Sherlock methodology is its focus on analysing variability and using the artefacts of that analysis throughout the whole process of SPL development. Sherlock addresses the problem of developing multiple products by grouping such products into product lines, according to an existing commonality among them and proposes a way to develop them in a coordinated way.

Sherlock can also be successfully used when developing single products. Sometimes this is the only feasible way to proceed. Consider a company, which decides to start developing software systems in a new domain. If the company has and is willing to invest a large amount of money, it can embark upon developing a series of systems from the very beginning. Taking into account that usually money is a scarce resource, this is seldom the case. Moreover, the domain can be undergoing a transition, so a "pilot project" can be used to acquire more experience and confidence about the specifics in that domain. Applying Sherlock in such an environment will be useful since it will necessitate the acquisition and analysis of vital information. Doing so would mitigate the risk of starting from a wrong point and would ensure that scalable design of the software system is delivered from the very start. Thus, the initial product, if successful, will be an advantage not a maintenance burden which repercussions should be dealt with for a long time.

Sherlock consists of five phases. A brief description of each phase is provided in Table 1. The phases should not be considered strictly consecutive. In fact, the sequence of execution can vary from company to company depending on the already existing software development practices. Therefore, Sherlock can be considered as an augmentation of a traditional software process with an emphasis on the domain.

Domain Definition	Defines the boundaries of the domain, collects and organizes appropriate information. Performs feasibility analysis that determines whether it is meaningful to continue with DAE.
Domain Characterization	Analyses the market conditions including existing products, user groups, and formats. On that basis it is determined what products should be developed and what features these products should implement in order to provide higher value for users.
Domain Scoping	Analyses commonality and variability among different products in a domain. Maps products on the variability space and presents strategies for developing these products regarding them as sets of variants.
Domain Modelling	Captures requirements from the analysis of the products in the domain. Present these requirements in the form of use cases and in object-oriented analysis modes. Use these representations to further identify commonality and variability.
Domain Framework Development	Design and develop a reusable framework for products in the domain. This framework comprises reusable components, which are used to facilitate the development of existing and future products.

Table 1: Sherlock Phases

Domain characterization, domain scooping, and domain modelling are the three core phases of SPL; this is where most decisions are taken. The three phases are inter-dependent; each phase uses information from the other phases, processes it, and makes it available to the other phases for further refinement.

Domain characterization, domain scoping, and domain modelling set up a loop responsible for the market and technical analysis of the domain. More specifically, the loop identifies, describes and evaluates the interesting products in the domain. Domain characterization provides the study of complementary and competing products in the market analysis, so that the products can be positioned properly in the market. Domain scoping manages the variability between the products. Domain modelling models the products from requirements perspective.

These three phases rely heavily on the concept of a "variation point". A variation point is a source of variability in the domain. It is a common feature implemented in different ways in different products or in a single product. Domain characterization and domain modelling find the variation points. Domain characterization seeks them in the market; domain modelling identifies them thorough a model of the products and of the domain. Domain scoping organizes the variation points creating a variability space on which choices and strategies can be visualized and evaluated.

The order in which we introduce these phases identifies start-to-start dependencies among them that we found in most applications of Sherlock. Domain characterization is a natural starting point: we look around for what is available. Domain scoping is the "variability centre" of a process based on the analysis of variability. Domain scoping is thus the fulcrum of DAE. Domain modelling deepens the initial model of the domain with a unique formalization. Domain framework development is the final phase in Sherlock in the sense that its deliverables are not used as input by the other phases.

The following five sections provide more in-depth description of the activities and artefacts included in each phase.

3.1 Domain Definition

Domain Definition's goal is to collect the necessary information that will help a software practitioner decide whether to continue with DAE and adopt a product line approach to developing a number of products. Common reasons why sometimes it is not beneficial to

adopt these practices are: the domain to be analysed is extremely unstable, the company does not have well defined business goal, or the cost of building reusable components won't be well amortized by the envisioned products. In order to make such a decision Domain Definition (DD) collect information from a variety of sources including market and domain experts, developers, etc.

Once iterated through DD produces the following artefacts:

- Domain terminology vocabulary
- Classified information on the domain
- Definitions of feasible, strategic, and current domain
- Feasibility analysis

The domain vocabulary is incrementally built during the phase of domain definition. The main reason for compiling this information is to facilitate the communication among all the parties involved in DAE while using the precise terms adopted in the domain. Naturally, modifications to the definitions of some terms can occur during later phases of DAE, thus, necessitating the propagation of such modifications to all documents that refer to these terms. This is not always possible because the documents may exist only in paper format or may belong to external organizations. In order to easily maintain the consistency between the domain vocabulary and all documents pertaining to DAE, the use of a specialized CASE tool is often required.

To make the information gathered from different sources truly useful, it needs to be classified in a way that makes referring to it easy. The classified information about the domain includes the following categories: user's perspective; market conditions and company's strategy; and information technology. Useful sources to explore include: history of the company, information from domain experts, information from market experts, and company's strategy.

In the process of collecting domain information, better understanding of its scope is achieved. This makes it possible to further classify the information, and consequently the

products that are associated with that information into three domains: current, feasible, and strategic. The current domain includes the products that are currently under development, both at the company that performs DAE and its competitors. The strategic domain includes the products that are in line with company's mission and targeted market niches. Consequently, this is the domain in which a company is most interested. The feasible domain includes all possible products that can be envisioned by the company.

The last step of DD is feasibility analysis; it concludes whether it is beneficial to continue with the subsequent phases. Even though, it is the last step in DD, feasibility analysis dominates to a large extent the whole phase and determines the focus of the information-gathering process. Independent of the outcome of feasibility analysis performing DD is always beneficial for a company. If the conclusion is that DAE should continue, than DD represents the first of its phases. In contrast, if feasibility analysis concludes that DAE is not proper for this domain, than the company has avoided making a substantial investment that later on will not be paid back.

3.2 Domain Characterization

Domain characterization is the observation of the domain and the planning of a strategy for the development of products in the domain. This observation regards the analysis of the products and firms in the domain, the pools of users, and the competitive or coordinating relations between the users and the products; also an explicit identification of the compatibility types in the domain is performed. All the products in the domain are grouped according to the type of compatibility elements they comply with. The planning regards the definition of a strategy that will produce profitable products. The goal can be to establish, take over, or maintain a market position in the domain. Any firm can benefit from domain characterization, although those who develop several products within a domain have more variables to control and thus, can achieve better results.

In domain characterization the products are characterized by their internal and external value. The internal value represents what the user perceives from the product alone, regardless of other existing products and the environment in which the product is used.

Conversely, external value is what the user perceives from the considered product in terms of ability to communicate with other users, share information, and interact with the environment as a whole. Hence, external value is highly dependent on the type of compatibility elements adopted by a product. Both internal and external value are represented by features, which express "what" the product should do and qualities, which express "how" the product should perform in terms of quality attributes the users are susceptive to.

The external value can be further decomposed into two aspects that are related to compatibility:

- Interoperability between users – users need to exchange information and they tend to attribute higher value to products that allow them to do so easily. In most cases the interoperability at this level is performed by exchanging information through files. This can roughly be defined as Input/Output compatibility.
- Interoperability between products – products interoperate with each other exchanging information through data files, networks, APIs, etc. As can be seen, the interoperability between users is a subset of the interoperability between products. The separation between these two is not clear-cut, but a rule of thumb is that in the former case we have explicit user actions to achieve the interoperability, as in the latter it can be done implicitly by the software system.

Often by using the interoperability between products more complex "aggregate" systems are constructed. For example, configuring a given IDE to use a Control Version System results in an aggregate system of two independent products that are used together by a particular user.

Two other terms that are used in conjunction with compatibility are installed base and network externalities. The installed base of a product is the pool of users using the product. Often the term "installed base" refers to the number of users of a given product. "Network externalities" is the term that expresses the increased value of a product as a result of its installed base. Network externalities are the underlying principle of the

external value. Due to compatibility, network externalities cross the installed bases of several products and hold at domain level. Often providing compatibility among the products in a domain can be expensive, if feasible at all. That is why we need to gather information about the installed bases of existing products in the domain and decide with which products it is most profitable to build compatibility with. Such a choice represents a trade-off between the expected effect of network externalities and the technical cost of implementing the compatibility.

Sometimes, market leaders can use proprietary formats that are not open to other companies, thus, intentionally limiting the chances for interoperability with other products. Such behaviour can be expected in relatively young domains in which there is no strong competition and a company holds a substantial part of the users in that domain. Compatibility is manifold and users can perceive products as compatible in many ways. Some examples of compatibility are the flowing:

- Compatibility of data formats (*.doc, *.pdf, *.ps, etc.)
- Compatibility of APIs and of protocols. (COM, DCOM, TCP/IP, HTTP, etc.)
- Compatibility of User Interfaces (MS Windows look-and-feel, Common Desktop Environment, etc.)

The list of compatibility types can include other domain specific categories that have to be identified by the software practitioner performing domain characterization. A compatibility element is an instance of one of the compatibility types, i.e. a compatibility element is a specific data format, API, protocol, UI element, etc. Products usually support a few compatibility elements from each compatibility type.

Two entities are responsible for specifying compatibility elements: firms and standardization organizations. Firms usually propose their own compatibility elements and have, from a technological and market viewpoint, full control over them. Sometimes firms coordinate with other firms with which they share the control of the compatibility elements. Standardization organizations are independent institutions, possibly supported

by governments or at international level; examples are ANSI, IEEE, etc. Standardization organizations may also originate as consortia between firms.

In order to provide interoperability among products that do not implement the same type of compatibility elements, converters can be used. The main function of converters can be expressed concisely by: providing compatibility between two (or more) incompatible formats. In order for a converter to be successful, its price should be lower then the expected cost of implementing the supported compatibility elements within a product.
Sometimes the lack of competitive standards in a given domain can create the, so-called, "bandwagon" effect. This effect is characterized by the fact that an influential organization in that domain adopts a standard (can be de-jure or de-facto). Other companies adopt the same standard, since they want to preserve compatibility with the leader and benefit from its large installed base. The earlier a company joins the leader, the higher are the benefits it can expect from its choice, mainly due to the experience and knowledge is manages to acquire before other companies join the network. But also, the earlier a company joins the leader, the worse is the damage if the network does not survive in the market.

3.3 Domain Scoping

Domain Scoping performs commonality and variability analysis of the products in the domain, which were identified in Domain Characterization. The goal of commonality analysis is to find common aspects shared by all the products in a domain. In order for a commonality to span a number of products it has to include a generality that leaves some aspects unspecified. These generalities in Domain Scoping are called variation points. Each variation point has a few different implementations that are called variants. A product can implement one or more variants per variation point.

Variation points come from variable parts in the domain. Products can 'vary' in many ways (Predonzani, et al., 2000):

- Several products share the same feature but implement it in different ways

- One product implements a feature in several ways and offers the user the choice between them
- One or more products implement a feature in the same way, but it is foreseeable that, in future products, the feature will be implemented in different ways.

Even though Domain Scoping is focused on variability, it does not search for variation points and variants. Rather, it relies on Domain Characterization and Domain Modelling for this search. Domain Characterization identifies the products in the market and produces a summary of their value. Domain Characterization does not look into product's implementation details; it focuses on compatibility and compatibility elements, which in most cases represent variation points. For example, output format can be a compatibility element for a number of products that provide user's interoperability. In addition, output format is a variation point with variants such as: plain text, "HTML", "RTF", etc. Domain Modelling, which will be described in the next sub-section, analyses a product from the technical perspective. Thus, the technical variability that arises from the many different ways in which a product can be implemented is identified.

Domain scoping organizes the variation points creating a variability space of the domain. Lets define VP to be the vector of all variation points, i.e. $VP = \{VP_1, VP_2, \ldots VP_n\}$, and V_i to be the vector of all variants for the variation point 'i', i.e. $V_i = \{ V_{i1}, V_{i2}, \ldots , V_{im}\}$. If we assume that a domain has n variation points, than the variability space will be defined by $V_1 * V_2 * \ldots * V_n$. Consequently, an element in that space will be represented by a n-element tuple, and the number of elements in the space will be m^n. For any practical problem the values of m and n will result in a variability space that will be very hard, if possible at all, to visualize and reason about. Therefore, the variability space is divided into regions that are analysed fairly independently. Each region includes only variation points that are correlated, i.e. the choice of a variant for one variation point depends to some extent on the choice of a variant belonging to another variation point.

The variability space can also be looked at as a map of different products. Each point on the variability space represents a specific choice of variants to be implemented. Since

products normally implement a few variants on a number of variation points, the products are often represented as areas on the variability space. Doing so facilitates their comparison and analysis. The variability space is also used as a container of strategies. A strategy is the idea or motivation behind the development of a product. Each strategy for a product is fully specified by the variants that are to be implemented by a product. Strategies normally result in different variations of a product such as: basic, advanced, enterprise, etc. The difference between strategies and products can be described by the following reasons. First, strategies are conceptual, while products are concrete entities, thus, a strategy can be applied to a number of different products that can even be competitors. Second, strategies concentrate on a much smaller subset of all variation points, as opposed to products, which normally have assigned variants for a large number of variation points existing in the domain.

To conclude, Domain Scoping bears many similarities to version management, but the activities and artefacts of Domain Scoping provide much stronger basis for product planning. Domain Scoping plans future products by devising product strategies based on the variability points in the domain. It uses the concept of variability space to visualize past, current, future, and competing products and thus facilitate the comparison among them. Domain Scoping can be divided into two main parts: first that explicitly defines and models the variability in the domain, and second that decides how the different products in the domain will map to this variability space in a way that will bring the biggest benefit to the company. A few heuristics that can be used to guide the process of choosing among different product strategies can be found in Predonzani et al. (2000). Domain Scoping is also the base of product lines: it provides the decision model for producing new products based on the common core and pool of variants.

3.4 Domain Modelling

Domain Modelling produces a model of the domain and a model for each product in the domain. The domain model is aimed at representing all the products in the domain and thus, needs to incorporate all the commonality among the products. Normally, the creation of the domain model starts with factoring out the commonality among a few

already existing product models. Thus, the domain model often looks like a generic product. The domain model is based on the same notation as the product model. Domain Modelling is based on the Unified Modelling Language (UML) (Booch et al., 1998) and more precisely consists of a use case model and an analysis model. The use case model defines use scenarios of a system from the perspective of a user; the analysis model describes the core functional and interfacing objects, and their relationships (Jacobson et al., 1997).

While the domain model is based on aggregating the common aspects of the products in a domain, it is not necessary to start its development after all product models are created. Usually, the domain model evolves as new products are added to the domain and the tendency is, as the domain model matures, each new product model to be created as a derivation of the domain model with few additions. In order to be easily customizable, the domain model expresses variability in terms of variation points and variants.

Domain Modelling comprises four types of models: product use case model(s), product analysis model(s), domain use case model, and domain analysis model. The commonality and variability within a product is expressed by connecting use cases by the "uses" and "extends" relationships. Similarly, the commonality and variability among the different products is represented in the domain use case model by using the same use case relationships. Each use case model has a corresponding analysis model, and they are connected to each other via a traceability relationship (Jacobson et al., 1997).

After a domain use case and analysis model is created, the creation of each new product starts from domain modelling and targets at mapping the new product as closely as possible to the existing domain models. Naturally, during that phase revisions to the domain model may be required, which determines the iterative nature of Domain Modelling.

3.5 Domain Framework Development

Domain Framework consists of a set of software components and an architecture designed to build the applications in the domain. In this case a software component denotes a piece of software that provides certain functionality and can be deployed individually. Software architecture denotes the structure of the applications in the domain and consequently specifies how the software components should be glued together. The domain software architecture consists of several "projections" which are views over the software structure from different perspectives. The domain framework consists of two layers: a layer specific to the operational environment, and a layer specific to the domain functionality.

The development of domain products benefits from the existence of the domain framework in the following manner: each product's functionality is mapped to existing software components from the domain framework, and these components are integrated following the structure of the domain architecture. This way of development ensures that all applications have consistent design, which has been validated and verified and benefit from software component reuse. As a result the maintenance of multiple products is alleviated and the effort of propagating bug fixes is significantly reduced. On the whole, the development of domain products by exploiting the domain framework artefacts is significantly faster, more cost effective, and easier.

In practice, the domain framework alone doesn't provide the possibility for developing complete products. Each product consists of software components reused from the domain framework and product specific code. Hence, one of the main difficulties in developing products is to incorporate these two parts into a cohesive software entity. There are several ways in which a framework and a product specific part can cooperate (Predonzani et al., 2000):

- The framework works as a box of components. The product specific part picks the right components and puts them together as is more appropriate for the product

- The framework provides the skeleton of a product. The product specific part provides the components. The resulting product has the shape of the skeleton with the substance of the product-specific components.
- The framework provides the skeleton and the components of a product. The product-specific part attaches the components to the skeleton. In this process, the product-specific part has a certain degree of choice, as some components in the framework are interchangeable.

The domain framework can provide functionality not only via software components that are used "as they are", but also through hooks for attaching behaviour. The same applies to the product specific part. Depending on the implementation, hooks can be of different kind including: an interface or an abstract class to be implemented, a concrete class that has to be subclassed and instantiated, and a pointer to a function that has to be associated with a real function. When building domain products, a software developer links the framework and the product-specific part by connecting the hooks these modules provide. On the other hand, a product specific part may comprise mostly of code that connects software components or implements hooks belonging to the framework.

In Sherlock two main types of frameworks are identified: passive and reactive. A passive framework doesn't impose a specific control flow on the applications that are built using it. Conversely, a reactive framework is usually based on callback functions and loops thus, significantly imposing its own control flow over the application that uses it. The choice of which type of framework will be developed depends on the similarities of domain products. If they can be organized around the same event-driven metaphor then developing a reactive framework is the best choice since it will insure adherence to this metaphor. On the other side, if the applications in the domain have significantly different non-functional requirements and are based on different control flow mechanisms than the domain framework should impose looser assumptions and thus avoid an architectural mismatch.

A number of studies have shown that a single view/paradigm for building a software architecture is rarely sufficient for developing well-modularized systems (Kruchten, 1995) (Tarr et al., 1999) (Kiczales et al., 1997). In Sherlock three projections on the software architecture are required: package projection, class projection, and object projection. Additionally, an object interaction projection, a functional projection, a process projection, and component projection can be developed. The three mandatory projections are presented in one or more UML diagrams of the corresponding type.

The package projection specifies how the design documents and the source code have been divided to ease the task of developers. This separation has important managerial implication since it allows the development team to be divided into several groups that work in parallel on a number of large-grained tasks. Packages can include other packages or depend on packages at the same or higher hierarchical level and such relations among them should be clearly denoted in the package projection. Reference cycles are allowed, but hierarchical structure cycles are not. The package projection is described in one or more UML package diagrams.

The class projection describes all the classes of objects in the system. The classes are connected to each other using association, aggregation, composition, generalization, etc. The classes in the class projection are directly related to the classes in the analysis model created in Domain Modelling, even though the correspondence is not one-to-one. In most cases a class from the analysis model is decomposed into a number of classes in the class projection. UML class diagram is the notation used for representing the class projection.

The object dynamic projection is very similar to the class projection but possesses a few characteristics that make it very important for obtaining a better understanding of the system. The objects represent the system at run-time as opposed to classes that represent the static structure of the system. Additionally, the number of classes in a system can be easily determined by inspecting the source code, but the number of objects at run-time can be very hard to determine because it may vary substantially during the execution of a system. Representing all possible configurations of objects at run-time is not practical for

most systems and consequently the object projection targets at capturing object interactions for typical execution scenarios that are of particular importance for the system. The object dynamics projection represents the sequence of messages exchanged among interacting objects in a given scenario without taking time into account. The object interaction diagram includes information about the timing and delays in message exchanges. Since this information is quite expensive to gather, an object interaction diagram is constructed only when time is a critical aspect of the system's functionality.

The functional projection divides the system into parts that perform clearly visible, distinct functions. The functional projection also describes the connections between the parts in terms of exchanged and shared data. The process projection describes the processes that comprise the system at run-time and the functionality that they perform. The component projection describes the major components in the framework, and can be used to describe the interactions between the running subsystems, from the point of view of components.

To successfully use the domain framework, in addition to defining the nature of the framework, i.e. reactive or passive and the different projections of the framework architecture, the following artefacts are required: documentation of the components in the framework, a browse able/searchable catalogue of the components in the framework, and guidelines for developing applications using the framework.

The documentation of a component consists of:

- A full description of the component in the package projection.
- A brief description of the functionality of the component – used by a developer to decide whether this is the component he/she needs
- A detailed description of the component, including references to the architectural documents related to the component, and typical examples of how to reuse the component

The choice of whether to implement a browseable or searchable catalogue of the components in the framework depends on its complexity. In relatively small frameworks, a browseable catalogue that has alphabetical ordering will suffice. In complex frameworks a more structured ordering of the catalogue and search capabilities are needed. According to Prieto-Diaz (1991) there are two widely used techniques for organizing catalogues: hierarchical and faceted. In the hierarchical approach the set of components split into categories; each category is further split into subcategories, and so on until at n-th level of the hierarchy contains components that are closely related. Consequently, browsing in this structure will have the properties of a tree search. The faceted approach acknowledges the fact that often it is hard to pick a single category a component belongs to. Hence, the faceted classification considers the different categories, called facets, separately. A user has to know or guess the relation of a component to one or more facets in order to perform a search for this component. The faceted approach is more expensive than the hierarchical since it requires the development of a search engine and the initial classification of components according to different facets.

The guidelines of how to use the framework to develop applications can significantly influence the success of the framework. Predonzani et al. (2000) recommends the following steps for developing the guidelines:

- Identify the main kinds of potential applications that can be developed using the framework. The list of products in the current domain is a good starting point.
- Provide examples of how to combine classes and objects to create such products. This refers to a form of reuse often called "black-box reuse" or "reuse as is".
- Discuss how to create new product-specific components based on the framework's components when simple composition of existing components is not sufficient. This refers to a form of reuse called "white-box reuse" or "reuse with change".

3.6 Summary

Sherlock consists of five phases, which incrementally collect information and represent our knowledge about the domain. Domain Definition (DD) compiles the most essential

information about the domain used to ascertain the feasibility and economic value of continuing with DAE. DD also sets a common base on which the communication among the different stakeholders can be carried out. Domain Characterization identifies the compatibility types in the domain. In order to achieve better understanding about the products in the domain, a product analysis is performed. This analysis decomposes the products into internal, external value, and quality attributes making it easier to identify the core set of features for the products in the domain. The installed base and the inherent network externalities define the products that are the most beneficial to build compatibility with. Domain Scoping (DS) performs commonality and variability analysis and thus serves as the basis for defining product families in the domain. DS also identifies strategies for combining the variations present in the domain in a way that will be the most economically profitable. Domain Modelling represents the variability from a more technical perspective and aims at developing domain architecture. Domain Framework Development builds the reusable components that will be used in products' construction.

Clearly, Sherlock does not impose any explicit requirements on a company's organizational structure. The practices that constitute the different phases can be introduced as an 'augmentation of an existing software process and they interfere little with the low-level development techniques. It is true that in these circumstances the management has less control over the process of product line development and large-scale reuse in general. This disadvantage is counterbalanced by the smaller initial investment and the lowered risk of undertaking a major organizational change that can severely impede the progress of projects under development.

4 Need and requirements for an SPL support tool

The main difficulty in implementing SPL development stems from the large number of activities, which have to be carried out, and the information associated with them, which has to be compiled and continuously modified. To address this difficulty, an SPL support tool should be able to track the different types of information related to the different activities involved in a particular SPL methodology. Since these activities are interrelated, the tool also has to track the dependencies between different types of data. SPL involves many different knowledge contributors (i.e. domain experts, economists, developers, etc.), hence, a suitable tool should also support multiple users simultaneously. On the whole, an SPL support tool needs to maintain both the data and change consistency between many different activities that can be performed in parallel. In order to alleviate the difficulty of comprehending relationships between multiple, interrelated activities, a support tool should also provide semantic support. A representation of the interactions occurring among the different knowledge contributors in an SPL development process and the beneficial role a support tool can have is depicted on Figure 7. In the remainder of this chapter, the following aspects of an SPL support tool will be examined: traceability, tool integration, and semantic support.

The problem of data and change consistency is also known as the problem of traceability. Traceability concerns have already been observed and analysed in the field of requirements engineering. Gotel (Gotel, 1994) defines traceability in requirements as the ability to describe and follow the life of a requirement in both a forward and backward direction through the whole system's lifecycle. Traceability support has already been recognized as being valuable for tool support for domain analysis methods. For example, (Griss et all., 1998) includes "Support for Traceability" as one of the key concepts for tool support for the FeatuRSEB domain analysis method.

(Domges and Pohl, 1998) describe seven key capabilities of existing requirements traceability environments. These capabilities are predefined and customizable data types, predefined and user-defined queries including filtering and sorting, comprehensive

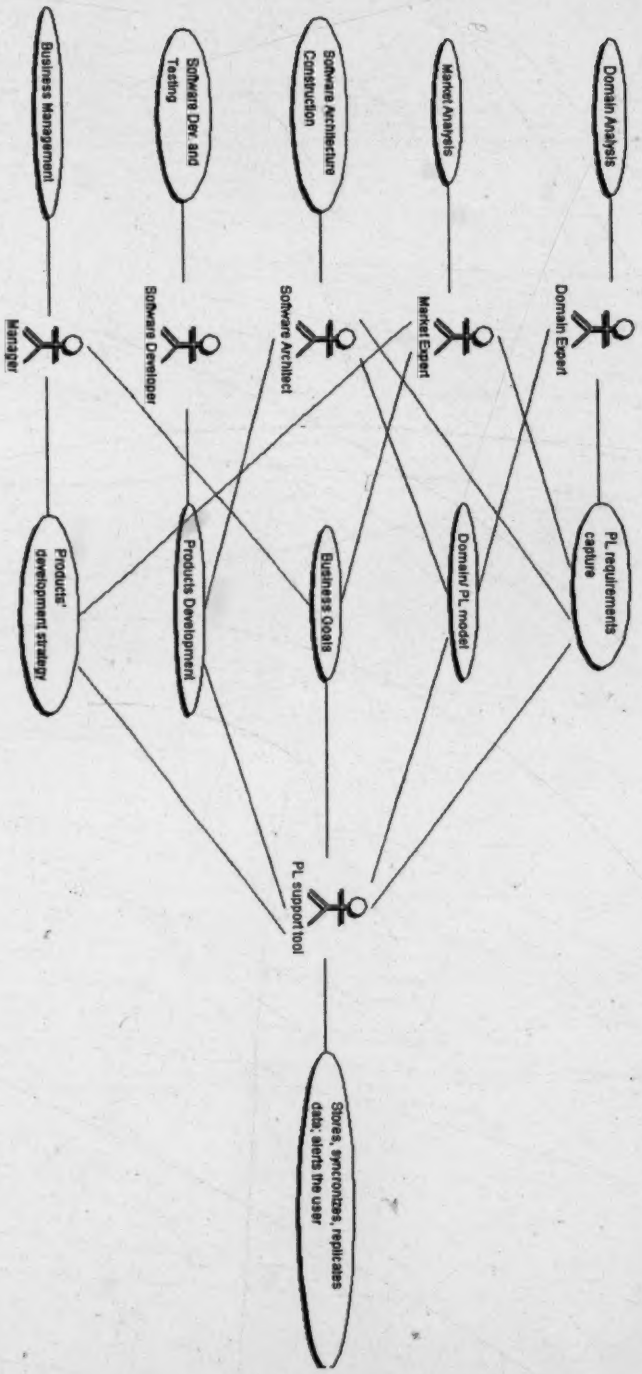


Figure 7: Use Case for PL support tool

configuration management and change tracking, trace analysis, various presentation formats, teamwork support, and interfaces to existing third-party software. These

capabilities seem to apply also to an SPL support tool with a few exceptions. SPL data types are likely to be predefined for a particular methodology and probably do not need to be customizable. Also having various presentation formats is not as essential as the other capabilities for SPL support.

The idea behind building an SPL support tool is to offer an integrated environment for developing, collecting, and retrieving the product line reusable artefacts. For each of the SPL specific activities, such as modelling or developing a domain vocabulary, there is often an already existing tool that can support it. It is preferable for the users to use existing tools rather than having to use a newly developed tool: users are already familiar with existing tools and specialized tools are more likely to have superior features. In addition, less time is needed to develop the SPL support tool, since the only effort needed is the integration of a tool into the overall system. An effective SPL tool should therefore be able to relatively easily interface to existing third-party applications.

Interfacing with third party software has already been analysed in the area of tool integration. Gautier (Gautier et al., 1995) identifies two aspects of software tool integration: tool-to-framework and tool-to-tool integration. Tool-to-framework integration suggests that the tools do not interface directly but instead use a framework to exchange data or operation calls. Conversely, when tool-to-tool integration is used a separate interface/adaptor should be created for every possible couple of cooperating tools.

Wasserman (Wasserman, 1989) describes four different dimensions of integration: user interface, data, control and process integration. User interface refers to a common "look-and-feel", data refers to the sharing of information, control refers to direct tool-to-tool communication, and process refers to tool activation based on a particular process model.

For the purpose of maintaining data and change consistency, the most important integration dimension is data. Data integration can be achieved through a shared repository or by direct data transfers. The other integration dimensions may be desirable

for a SPL tool, but not essential. Even considering data integration on its own, tool integration is difficult to achieve because "in general, domain information is stored in a great variety of data sources, using different data models, access mechanisms, and platforms." (Braga et. al., 1999) In this light, tool-to-framework integration seems more appropriate as it removes the problems of requiring combinatorial adapter interfaces between multiple third-party tools.

The complexity and the variety of activities involved in SPL development suggests that some form of assistance would be extremely valuable and should focus on the semantic relationships between data rather than just syntactic checking. The concept of design critics suggests a suitable approach for semantic assistance. Design critics are intelligent mechanisms that analyse a design and provide feedback to assist the designer in improving it (Robbins, 1998). According to Fischer (Fischer et al., 1993), design critics should be embedded in the environment and actively but non-disruptively alert designer of potential problems suggesting potential solutions, if possible. (Robbins and Redmiles, 1998) described the critiquing approach as different from traditional software analysis approaches in that the focus is on a designer's cognitive needs. Traditional approaches attempt to prove correctness of a completed or nearly completed system. Critiquing approaches, on the other hand, pessimistically detect potential problems in partially specified systems. The P3, domain-specific component generator (Batory et. al., 2000) uses a tool called a design wizard, which is somewhat similar to design critics except that it is targeted toward optimizing data structures rather than higher-level design advice.

The presented analysis suggests that a SPL tool should support the following activities:

- Link consistency management: ensures that link traces make sense
- Change consistency management between different activities: ensures that all changes are propagated correctly
- Simultaneous multiple users: allows more than one user on the system at once
- Integration with COTS tools: allows COTS tools to communicate with the framework
- Semantic assistance: warns user of potential problems.

There are already existing tools to support Domain Engineering. These tools can be successfully used to a large extent in software product line development. Examples of such tools are Metaprogramming Text Processor (Prosperity Heights Software, 1999), EDGE (Loral Defense Systems, 1996), Diversity/CDA (Bayer et al., 1999A), Odyssey/DE (Braga et al., 1999), DARE-COTS (Frakes et al., 1998), DADSE (Terry et al., 1995) and DOMAIN (Tracz and Coglianese, 1995).

A limitation of existing tools is that some focus mainly on the late phases of domain engineering such as domain modelling and framework development. The practices that produce the requirements documentation for the product line and analyse the relationship among the products and the users in the domain are often missing in the existing support environments. In addition, their support of queries, change consistency management, multiple users, and semantic support, is often inadequate (Table 2).

Tool	Management of early phases	Link consistency management	Change consistency management	Multi-user support	Tool integration support	Semantic support
DADSE		No explicit links		Simple configuration management with file locking	Tcl/Tk scripting for tools not needing access to system data	Knowledge-based design assistants
DARE-COTS		Manual; links represented in "Domain Book"			Loose integration; all tools are COTS	
Diversity/ CDA		Textual and graphical link browsers	Uses InfoBus for change propagation	Long-term locking for multi-user consistency	Supports COTS tools with InfoBus interface(s)	
DOMAIN		Chimera hyperweb browser			Interfaces with tools supplying a Chimera-compliant interface.	

[ ] - Not/Poorly Supported [Text] - Details

Table 2: Mapping of existing tools to identified requirements

5 Holmes – a support tool for SPL development

The prototype of an SPL support tool, developed as a main subject of this thesis, is called Holmes. Holmes uses a blackboard architecture (Figure 8) built using a tuple space (Gelernter et al. 1985), as implemented in Sun's JavaSpaces (Freeman et al., 1999). In the next subsection a description of the characteristics of the Blackboard approach, as defined by Corkill (Corkill, 1991), and Holmes' mapping to them is provided. The section continues with further discussion of the benefits and issues encountered when using JavaSpaces as a communication medium, the persistent storage in Holmes, and the critiquing system.

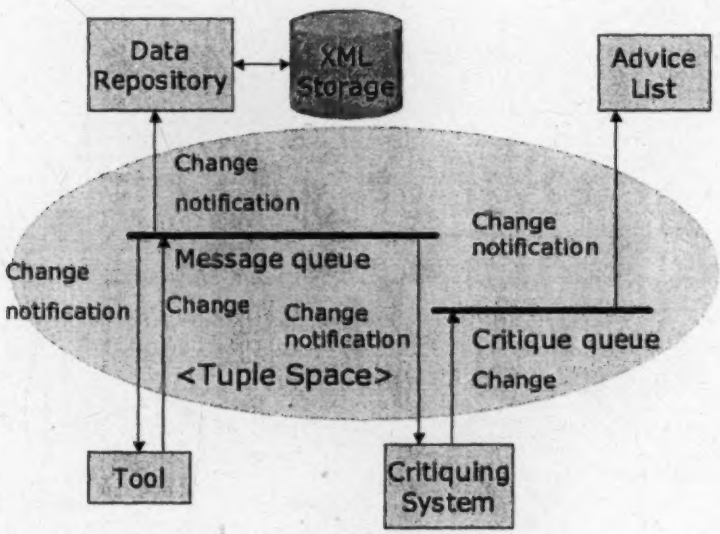


Figure 8: Architecture

5.1 Blackboard Systems

The blackboard approach has been designed to address the problem of developing applications that solve complex problems and have ill-defined specification. Blackboard systems comprise a common data repository and a number of program entities that can

manipulate the data in this repository while working towards a common goal. Each program entity represents specific expertise that is helpful in solving the problem addressed by the blackboard system. Corkill (Corkill, 1991) identifies eight main characteristics of Blackboard systems:

- Independence of expertise
- Diversity in problem-solving techniques
- Flexible representation of blackboard information
- Common interaction language
- Positioning metrics
- Event-based activation
- Need for control
- Incremental solution generation

Each program entity in the Blackboard system, often called knowledge source (KS), represents self-contained expertise that can be used in solving a problem addressed by the application, i.e., no KS requires other KSs in order to apply its knowledge and to make its contribution. Once a KS identifies information in the Blackboard that is meaningful to it, the KS proceeds without any communication with other KSs. In this way, it represents a human expert that works fairly independently together with a group of other experts towards solving a common problem. The independence among the different KS makes it easy to extend and enhance the Blackboard system incrementally: at any time one or more KS can be modified, removed, or substituted with superior one without affecting the rest of the system. Because in most cases the KS represent relatively large program entities compared to, say, a set of closely related rules (as in Expert Systems), the Blackboard systems differ from other AI problem-solving techniques with their coarse-grained modularity, which facilitates continuous development of the system.

Analogously in Holmes, each tool can exist and be used independently from the rest. Holmes comprises five main tools that correspond to each phase of Sherlock. In addition, some of these tools consist of a number of other tools that correspond to the different activities performed in the associated phase. For example, the Domain Definition tool

provides separate tools for modifying the domain vocabulary, the classified information, the structure of the domain, and the feasibility analysis. Even the tools that constitute the Domain Definition tool communicate among each other through the blackboard. Thus, each tool is completely independent from the other tools and any modifications or extensions to the system will be transparent as long as the information posted by the new/modified tools is meaningful.

The Blackboard architecture provides for diversity in the problem-solving techniques used by the different KS. Since each KS does not depend on any other one, the details of a KS inference technique are completely hidden from the other KS. This loose coupling among the KS makes it possible to combine completely different approaches to solving a given problem: rule-based systems, neural networks, fuzzy-logic modules, linear-programming algorithms, etc.

With the exception of the critiquing system, Holmes' tools do not employ such AI mechanisms, but there is a significant diversity in the implementation of the incorporated tools – some are third-party, some are internally developed. Also, the type of information that the tools manipulate and the type of experts that use them differ considerably. All of the tools except for the critiquing system, which will be described in section 6.5, are meant to provide automating rather than problem-solving support. As such the tools in the first two phases, namely Domain Definition and Domain Characterization, deal mostly with textual information. The experts that are involved in these phases are domain experts and market experts. Conversely, Domain Scoping and Domain Modelling are used to manipulate mostly graphical information that represents the artefacts of variability analysis and the design models in the domain. Hence, the experts involved are software architects and project leaders. Domain Framework Development is concerned with building the reusable framework that will be used to produce the products in the domain and consequently consists of source code. Its user group is software developers. As the knowledge in the above-mentioned areas matures, and the use of artificial intelligence techniques becomes more viable, their integration will be seamless and will not affect the rest of the system.

Given the diversity of knowledge contributors in the Blackboard approach, it is necessary to allow flexible representation of information. There are no prior constraints or rules about how the information should be put on the blackboard. Some KS might impose consistency on the information they post; others might put incompatible alternatives in order to improve the possibility of exploring a larger number of possible solutions. Although, considerable freedom in the information representation is allowed, all KS should adhere to some sort of common interaction language. Thus, meaningful interaction among the KS is guaranteed by correctly interpreting the information posted on the blackboard. Often, it is hard to strike the balance between an expressive language that is understood only by few KS and more generalized language that is understood by all KS participating in the blackboard system.

Hierarchy of command classes governs the information representation in Holmes. When a tool posts information on the blackboard, it has to be a subclass of the root command class. Beyond the type constraints imposed by the different commands, the values of different fields in a command object are not strictly specified. The fact that each command is an object, with the inherent polymorphic properties, allows for considerable flexibility in information representation. Thus, if a tool needs to extend or modify the behaviour of a specific command a new subclass with the desired behaviour can be created. Following the Liskov substitution principle (Liskov, 1988) the rest of the tools will treat this new type of commands in the same way as the already existing ones.

The growing amount of data on the blackboard makes it necessary to provide some mechanism for locating relevant information. A common approach is to divide the blackboard into regions that contain semantically related information. Thus, a KS can monitor only the regions of the blackboard that it is interested in. To further facilitate locating pertinent information, positioning metrics can be used within each region of the blackboard. The ordering schema can employ different criteria such as: temporal ordering, relevance, size, or any other measure meaningful to the posted data. As a result, a KS not only finds information faster but also may infer additional information from the

applied ordering. The amount of data stored on the blackboard will grow indefinitely, unless some mechanism for removing outdated or unnecessary information is provided. A garbage collector can be also based on the positioning metrics, sometimes introducing new ordering criteria that will better represent data not valuable to the decision process.

In order to address the problem of locating pertinent information, the blackboard in Holmes is divided into conceptual queues. Each queue contains semantically related information. Since in JavaSpaces there is no build-in mechanism for structuring the information in the tuple space, the queues are defined on the basis of class types. For example, all commands that represent activities performed on the domain vocabulary inherit from *DomainTermCommand* class. Even though there are no explicitly defined regions on the blackboard, by committing themselves to particular type(s) of commands the tools don't need to examine every command posted on the blackboard. To further improve the ability for locating relevant information within a given queue, Holmes implements the notion of positioning metrics by assigning an index on each command. The index number also reveals a temporal relationship among the commands, i.e. if a command A has an index smaller than command B, then A has been posted on the blackboard before B. The amount of information on the blackboard is kept from growing indefinitely by utilizing the JavaSpaces mechanism of time leases. Each command is specified a period after which it is automatically garbage collected.

The communication among the loosely coupled KS is accomplished through event-based activation. Each KS is activated in response to a new post that is relevant to this KS. In addition, a KS can respond to events that have occurred outside the blackboard. In the former cases the blackboard system should provide a mechanism for a KS to register for the type of events it wants to be notified for. The events occurring on the blackboard are: addition of new information, modification of existing information, or removal of irrelevant information. Some of the KS can communicate with the operating environment in which the blackboard system executes, for example, storing data persistently in files, handling network communication, etc. If any of these events are related to the decision process than the corresponding KS should translate the event into meaningful post to the

blackboard. In this way, most of KSs remain decoupled from the operating environment and consequently simpler and easier to port.

In Holmes, the tools collaborate by posting commands and responding to commands posted by other tools. Each tool has associated reader(s) that monitor the queue(s) in which the tool is interested. Since the readers are separate threads, the tool can be idle, perform computational tasks, or respond to user actions until a new command that contains relevant information is posted on the blackboard. For each queue that a tool is interested in, the tool has to create a separate reader and, thus, it explicitly "registers" for the commands that are posted on that queue. The tools read and write, i.e., add commands, but they never delete a command that has been already posted. This mechanism removes the possibility of certain race conditions, and even though results in more verbose communication, it ensures robust execution. So far, no event from the operational environment has been identified as meaningful to the operation of the system.

To ensure some order in the way contributions are made by the different KS in a blackboard system, some type of control is needed. Consequently, a control component that is separate from all the other KS and is responsible for managing the process of decision-making should be developed. The main objective of the control component is to evaluate in advance the benefit of prospective KSs contributions, and according to these estimations the control component will activate the KS with the most valuable expected contribution. In order to produce such estimations, each KS should be able to evaluate the resources needed and the quality of its contribution and then provide the results to the control component. In that case, the control component does not need to duplicate any of the expertise of the KSs and remains independent of the number of KSs in the system and their inner details.

In Holmes, a control component that manages the execution of the other tools does not exist. The reason for this is mainly due to the strict definition of each phase in Sherlock and consequently the lack of ambiguity when using the associated tool in Holmes. A

problem-solving support is provided to the user via the critiquing system, which analyses the proposed solutions for possible errors.

The decision-making process within a blackboard system proceeds incrementally. Different KS contribute to the current state of a solution by refining, contradicting, or starting a new line of reasoning. The sequence in which the KS make their contributions is not predefined and can differ from execution to execution. Thus, blackboard systems are particularly effective when the solution includes many stages and allows a few potential ways to proceed from a given stage.

Even though the different phases in Sherlock do not follow a strict sequence, recommended patterns in which the different phases are instantiated were provided in Section 3. The process of DAE and SPL are both incremental and iterative processes in which many knowledge contributors participate to a varying degree. All that makes the Blackboard approach an appropriate choice both from a technical and problem-solving perspective.

5.2 Tuple space

A tuple space is a form of virtual, shared, associative memory that generatively provides a repository for tuples. One of its characteristic features is that an entity, which is put in the tuple space, continues to reside there as long as some other process doesn't take it from the space. The existence of an entity in the tuple space is completely decoupled from the existence of the process that created that entity and put it in the space. That way a tuple space can be used as a storage that holds data to be exchanged among processes whose execution should not be constrained to overlap in time. Another advantage of using an implementation of a tuple space is that it provides for an easy way to synchronize a number of processes, which try to manipulate a set of objects. In that case a process is given a random object among those that meat the process' criteria. After the object is processed it can be put back in the tuple space or disposed of depending on the need.

In addition to the time decoupling, a tuple space provides also space decoupling. In the JavaSpaces implementation this is realized by using Jini distributed technology (Sun Microsystems, Inc, 2001). That way every process which wants to access a JavaSpace uses the same uniform way of connecting to it independent of the fact whether the JavaSpace is executed on the same machine or somewhere over a network.

All that considered the use of JavaSpaces gives an opportunity to implement a loose tool-to-framework integration that proved to be a better choice concerning the diversity and the number of the tools incorporated. Furthermore, the access method to any entity stored in JavaSpaces eliminates most of the difficulties usually associated with multiple users accessing a common resource. Certainly, such a method will not be acceptable for any time-critical application but real-time response is not an issue for an SPL support environment.

The different tools incorporated in Holmes communicate using an algorithm similar to message based communication. In the proposed implementation there are a few improvements over standard message passing. As mentioned above, the use of JavaSpaces waives the difficulty of initialising properly the communication channels between tools, after they are started. JavaSpaces stores messages as long as it is specified including infinity. Other improvements are that the messages that are exchanged among the tools are objects not data; a message queue implemented herein is not susceptible to the number of tools that "subscribe" to the data posted on that queue.

The advantage of using objects, compared to simple data, as messages is that an object encapsulates certain functionality. Thus, the tools that communicate through objects are decoupled from the format of the data stored in the object, which actually represents a message. This flexibility proves to be essential when dealing with diverse third-party tools, which use different output formats. Implementing messages as objects, in fact, turns them into functional units that perform the communication protocol. This way, changes made to the communication protocol leave the tools intact. The opportunity of

using object-oriented design patterns, such as Adaptor, Visitor, Strategy (Gamma et al., 1995), leave space for broad future evolution of the behaviour offered by messages based on objects.

Even though there are no clearly defined communication channels for exchanging messages among tools, the objects, which represent these messages, are organized into conceptual queues. The idea behind these queues is that they group messages belonging to semantically related data. Considering this organization, a tool subscribes to any queue that represents a flow of data the tool is interested in. A tool can act as both information provider and information subscriber. To do so, it has to instantiate an object of type "writer" and "reader" respectively. All readers of a certain queue monitor the JavaSpace for specific types of messages. The order in which the readers subscribed to a queue is unrelated to the order in which they will actually read a message. In fact, that order is unpredictable and varies from message to message. Nevertheless, the sequence in which messages are posted to the queue is preserved when they are read. (if a message M1 is posted to a given queue before message M2, than M1 will be read by a given reader before M2)

Tools that act as information subscribers are not aware and are not dependent on the type of tools that act as information providers, as long as they post messages that conform to the agreed interface. As described so far, the framework, used to integrate all the tools provides for loose coupling among the different tools in terms of time, type, and space. One disadvantage of the current framework is the possible performance loss compared to having a framework that supports individual message channels among the tools.

5.3 Persistent Storage

Holmes stores persistently its data through a number of tools that are called repositories. Every repository stores semantically related information just like queues group messages belonging to semantically related information. When the Holmes tool is started all other tools that are interested in particular type of information post a request for that

information. If the repository responsible for that information is started and has activated its reader than it will notice the request, as all other tools that monitor the queue, and will reply posting the required data. Similarly, when a tool makes a change to a certain piece of data, it posts the change on the queue. Doing so gives a chance to all other tools, including the repository tool, to update their copies as well. When the Holmes application is terminated only the data contained in the repository tools is saved to persistent storage.

The above-given way of dynamically storing data ensures that the repositories itself could hardly become a bottleneck in the system. The data or parts of it is distributed and duplicated among the tools that manipulate it. As shown, the use of JavaSpaces and objects as messages provides for parallel processing and polymorphic behaviour on behalf of the tools.

The choice of how Holmes data is statically stored is also related to the tool integration requirement. Holmes data is stored using the Holmes Markup Language (HML), which is essentially XML (W3C, 1998) with a custom Document Type Declaration (DTD). The advantage of an XML-based format is that the data can be viewed in a human-readable form using a text or XML viewer. This maintains independence of the data from the particular tool that manipulates it. The human-readable structure also provides the capability of building DTD translators to convert from HML to other XML-based languages. Although it does not completely eliminate the need to build adapters, in this case DTD translators, the effort is somewhat standardized and reduced. Obviously, the advantages of this approach rely on the growing popularity of XML as a data format, especially for Product Data Management systems and UML CASE tools.

5.4 Critiquing System

As pointed out in Chapter 4 some sort of semantic assistance would be highly valuable in an SPL support tool. The concept of a critiquing system suggests a suitable approach for such assistance. According to Ficher et al. (Fischer et al., 1991), "critiquing is the presentation of a reasoned opinion about a product or action". Thus, critics do not

necessarily provide a user with a solution. Their core task is to recognize and communicate debatable issues concerning an artefact. Critics point out errors and suboptimal conditions that might otherwise remain undetected by the expert. Critics can also advise users on how to improve an artefact and explain their reasoning. Consequently, critics help users avoid problems and learn different views and opinions.

5.4.1 Rationale

Robbins (Robbins, 1998) identifies the following reasons for including critics in a design tool: designers' limited domain knowledge, low relative cost of immediate revision, continuous learning, cost of failures arising from design errors, time-to-market, and risk management.

It is infeasible to expect a single designer to possess all the necessary knowledge in a complex domain; instead, large systems are designed by a team of experts working together. This is especially true in the DAE where multiple knowledge contributors cooperate in producing the necessary artefacts. Often, some of these domain experts are temporarily hired on site, and it can be complicated to consult them if a later review identifies unclear issues. Thus, getting timely revisions of external experts' work is essential for utilizing their expertise.

Most of the technologies in software development undergo constant change; hence, software practitioners have to frequently learn new skills and methodologies. A design critic can address this issue by providing explanations to identified problems. It is commonly recognized today that errors at early development lifecycle phases are much more expensive to fix if identified during implementation or testing as opposed to the phase in which they originate. A design critic can significantly lower the occurrence of some typical design faults that, if left unnoticed, can have costly repercussions. In addition, a design critic can substitute some of the activities normally performed during design reviews, thus, giving a possibility to shorten the phase of design as a whole, resulting in a faster time-to-market.

The main distinction between the critiquing approach and other more traditional approaches to design evaluation is that the feedback is provided while the designer is still working on the problem being critiqued. This results in improved designer's task performance, since the designer has in its short-term memory all the details that have influenced him to make a particular decision. Consequently, he or she is able to evaluate the provided critique better and faster applying the appropriate modifications, if necessary. On the other side, the feedback from the critiquing system should be provided in a non-disruptive way, i.e., the expert should have the choice of when and how often to consult the critique.

Another advantage of the critiquing approach is that it is based on the "informative assumption" (Robbins, 1998) that means that a critiquing system doesn't try to prove the correctness or presence of errors in a given design. Rather, its goal is to inform the designer of potential problems, possible corrective actions, and pending decisions. The smaller scope of the analysis performed by a critiquing system makes it possible to apply this approach on partial solutions. Doing so is an essential advantage compared to other design analysis approaches that require a more or less completed solution before they can be applied.

Critiquing systems are most suitable for problem domains that have the following properties (Fisher, 1993):

- Knowledge about the domain is incomplete and evolving
- The problem requirements can be specified only partially
- Necessary domain knowledge is distributed among many knowledge contributors

These properties map to Software Product Line development very well. The knowledge about the product line is incomplete at the outset of the development. The phase of Domain Engineering somewhat alleviates this problem, but many issues are left open, including the exact number of product line members that will be produced by utilizing the product line environment. As new product line members are developed, the core assets of the PL evolve. The second property of partially specified requirements is well represented

in SPL development: only a small part of the products that are intended to be produced by utilizing the core assets of the PL can be well specified at the beginning of the process. As described in Chapters 2 and 3, SPL development includes the cooperation of different experts – ranging from domain experts and market specialists to software developers. All this makes the development of SPL a very suitable domain for providing semantic support through a critiquing system.

5.4.2 Holmes' Implementation

The critiquing system is implemented in Holmes similarly to the other tools (Figure 9). It subscribes for particular type of information by activating a reader to the associated queue. The critiquing system can be configured to monitor the data flow only on specific queues, which are of interest, or to monitor the data flow on all queues Figure 10. This follows the first step of Activation in the ADAIR Critiquing Process (Robbins, 1998), which stipulates that only a subset of critics relevant to the current task should be active. In Holmes, there are two options: either the critics for each phase of DAE are active, or only the subset of critics that correspond to analysed phases are active, i.e. the phases for which there is an executing tool. The critiquing system bases its responses on the data objects that it intercepts on a given queue without needing any prior knowledge about the tool that placed the data objects. The advantage of this approach is that the critiquing system is completely decoupled from the tools that post data to the JavaSpaces. This decoupling includes both the type and the number of the tools.

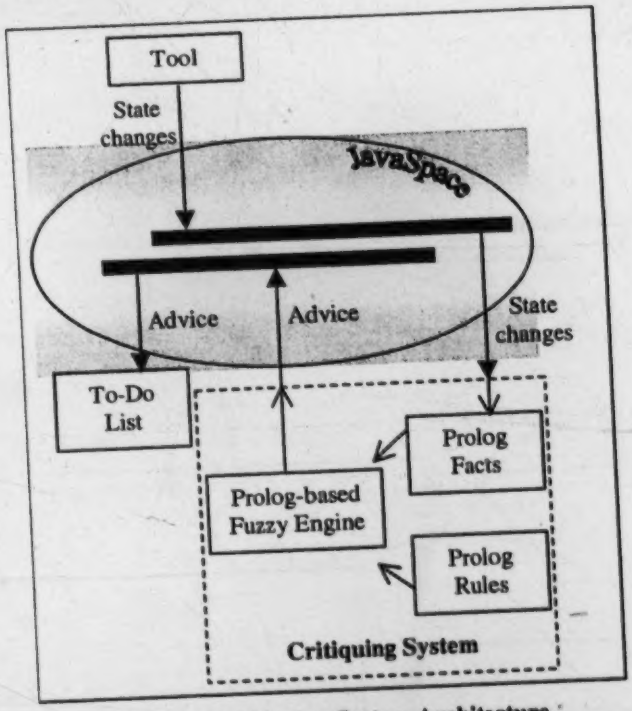


Figure 9: Critiquing System Architecture

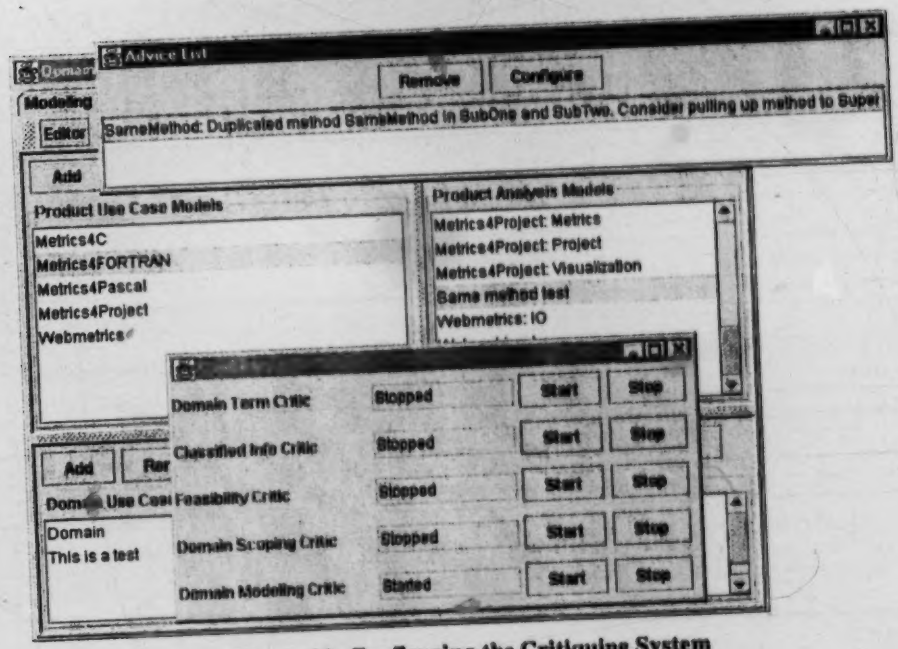


Figure 10: Configuring the Critiquing System

The critiquing system is based on the Prolog programming language. The knowledge about a specific field is represented in terms of Prolog clauses. Prolog was chosen, since its clauses are very well suited for description of relationships. Being a high-level language, Prolog is relatively easy to understand and consequently makes the process of evolving the domain knowledge plausible even for non-programmers. Critics can be triggered when a particular content type is posted to the JavaSpace; the semantic meaning of the data is analysed through the existing Prolog database, which represents the current knowledge about the domain. If any critique clauses are satisfied then a message, which describes the possible problem, is posted back to the JavaSpaces on a separate queue. Another tool which is called "Advice List" monitors that queue and displays the information contained from a given critique in a user-friendly way (Figure 10). The generation of critiques is based on user-modifiable rules written in Prolog. Since Prolog offers a possibility for dynamical assertion or retraction of clauses, the quality of the advice provided can be incrementally enhanced.

In order to integrate a Prolog engine with the rest of the system, which is developed in Java, the JPL (Dushin, 2000) bridge was used. JPL is a collection of Java classes and C functions that provide an interface between Java and Prolog. JPL uses Java Native Interface to connect to a Prolog engine through the Prolog Foreign Language Interface, which is gradually being standardized in different implementations of Prolog. JPL version 1.0.1 can interact only with the SWI-Prolog, version 3.1.0 or later (SWI-Prolog, 1987) and supports only the invocation of Prolog clauses and predicates within a Java program. The reverse direction of making calls to Java modules from Prolog is not implemented.

There are two levels on which JPL can be used – low level and high level. The low level is implemented in C and offers a flexible interface through a number of functions. The higher level is implemented in Java and even though is less flexible, hides most of the unnecessary implementation details from the user and is easier to use. In the next couple of paragraphs an overview of the high-level, Java interface is provided.

The class hierarchy of the JPL Java interface is represented by three root classes Figure 11: *Term, Query* and *JPLException*. The class *Term* is abstract and cannot be instantiated. A *Query* class contains a *Term* object that denotes the goal to be proven. The *Compound* class inherits from *Term* and contains an array of *Term* objects that represent the structure of the goal. At least one of these should be instantiated.

JPLException class has two children: *QueryInProgressException* and *PrologException*.

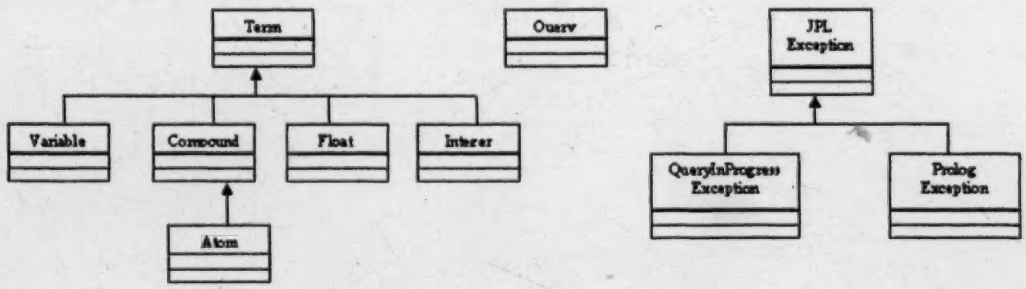


Figure 11: Class Hierarchy

There are several ways to initialise JPL. Automatic initialisation is most frequently used. It occurs when the first attempt to use the Prolog engine is made, that is, when the first *Query* object is instantiated. The default values for the command-line arguments are used for the Prolog engine. The class JPL provides methods for customized initialisation of the SWI-Prolog engine.

```
public String[] getDefaultInitArgs();
public void setDefaultInitArgs(String[] args);
```

The current version of JPL supports a single instantiation of the Prolog engine and doesn't offer a possibility for reinitiating it.

The Term-based classes in the High-Level Interface are best thought of as a structured concrete syntax for Prolog terms: they do not correspond to any particular terms within the Prolog engine; rather, they are the means for constructing queries which can be called within Prolog, and they are also the means for representing the results of such calls. Variables in Prolog are represented by a special class in JPL. The class *Variable* doesn't

have a property name, although it has a unique identifier. Reusing variable instances should be done only when the effects of doing so are well considered. The *Compound* class consists of a name and an array of *Terms*. It is the most often used class from the JPL hierarchy. All necessary parameters for a *Compound* object can be passed to its constructor Figure 12.

Java

```
Compound domainTerm=new Compound(
        "domainTerm",
        new Term[] {
         new Atom("CBO"),
         new Atom("Coupling Between
  Objects metric "),
        }
)
```

Prolog

```
domainTerm('CBO', 'Coupling Between
        Objects metric').
```

Figure 12: Example of a rule definition in JPL vs. Prolog

Every compound should be given a name that is denoted by the first parameter passed to the constructor. The next parameter is an array of *Terms*, which represent the arguments of that Prolog clause. Since *Compound* is inherits from *Term*, it can contain other *Compounds*, thus enabling the creation of recursive structures.

Constructing queries against a Prolog database is done through the use of the class *Query*. As mentioned before it requires an instantiated object of type *Compound*, which represents the goal to be satisfied.

```
Term goal = new Compound( "domainTerm",
        new Term[] {
                new Atom("CBO"),
                new Atom("Coupling Between Objects metric "),
        }
);
Query   q = new Query(goal);
```

The initialisation and skeleton of each critic execution is provided below.

	Java	Prolog
Initialize Prolog Engine	*JPL.init()*	
Consult a file	*Query query = new Query("consult", new Term[] {new Atom(fileName)});* *query.hasSolution();*	*?-consult(test.pl).*
Construct a goal.	*Compound goal = new Compound("man", new Term[] {new Atom("john")});*	*?-man(john).*
Construct a query	*Query query = new Query(goal);*	
Execute a query	*query.hasSolution();*	
Check more than one solution	*query.allSolutions();*	

Table 3: Critique Engine Initialization

Initialise the Prolog Engine. This is an optional step that can be skipped. The SWI-Prolog engine will be initialised automatically when the first call to a Query object is made.

Execute a query and check the solutions. When a query is executed there are three possibilities to check for solutions. If we are interested only whether a goal can be satisfied or not the most appropriate method of the *Query* class is: *public Boolean hasSolution()*. It returns "true" or "false" depending on whether the goal was satisfied or not. If there are a number of possible solutions but we are interested in only one of them the method *public Hashtable oneSolution()* returns a Hashtable containing one entry that represents the binding between the *Variable* and the *Term*. Conversely, if we want to manipulate all the solutions the method *public Hashtable allSolutions()* should be used. The Hashtable will be populated with the bindings of the variables present in the goal and their associated terms that satisfy the goal.

5.4.3 Problems encountered while using JPL.

One of the most serious problems encountered during development is adhering to the required hierarchal structure of Compounds when constructing complex Prolog clauses. More specifically, clauses have to be transformed from the syntax:

component1, functor, component2 [functor, component....]
*ex.: a+b*c*

to

functor,component1, component2,[functor, component1, component2]
ex.: +(a,(b,c))*

Figure 13 represents one such case.

Java

```
Variable X = new Variable();
Compound myRule = new Compound(
        ":-",
        new Term[] {
          new Compound(
                "development",
                new Term[] {X}
                ),
          new Compound(
                ",",
                new Term[] {
                  new Compound(
                        "design",
                        new Term[] {X}
                       ),
                  new Compound(
                        "implementation",
                        new Term[] {X}
                       )
```

Prolog

```
development(X) :- design(X),
        implementation(X).
```

```
))) );
```

Figure 13: Compound Structure

As it is obvious from the example (Figure 13) the corresponding Java program code is much more complex than the original Prolog code. This disparity proved to be quite troublesome when asserting or retracting existing Prolog clauses. In this case, the clause to be asserted should be parsed, so that the different variables and subgoals that constitute the clause are identified. After the parsing step, every element of the original clause is wrapped in the associated JPL class. Implementing a Prolog parser from scratch requires substantial effort, so that other ways to implement the same functionality were sought.

One possible way to overcome this problem is to use the Prolog build-in predicate *consult()*. In that case all the Prolog clauses that will be added should be first written to a file. That solution is applicable only if we have all the rules to be asserted in advance.
On the other hand, if the clauses to be asserted depend on the program flow, then such a solution would be inappropriate, because a temporary file for every Prolog clause should be created. SWI-Prolog provides a build-in predicate for transforming an Atom to a Term. The predicate *atom_to_term()* receives the following number of parameters:

- An atom that represents the clause to be asserted
- A term which will hold the transformed clause
- A list that will hold the bindings of the actual variables in the clause.

By using that build-in predicate one can construct a Prolog rule that transforms and asserts a Prolog clause without expending any effort on the necessary transformation.

dynamicAdd(X) :- atom_to_term(X,Y,_), assert(Y).

5.5 Sample Session

In this section a sample session of using Holmes is presented. The activities constituting Sherlock (Chapter 3) are now performed with the help of Holmes. The section presents

the functionality of Holmes and reveals the benefits of using automated support through the process of SPL development and more precisely the phase of Domain Engineering.

The five phases in Sherlock have corresponding tools. Each of these tools can be executed on a different machine, and as long as the support environment (Holmes) on these machines is configured to connect to a single tuple space, all of these tools will see the same information and the data consistency will be maintained. When starting a new project the first tool that is normally invoked in Holmes is domain definition. The activities supported by this tool are compiling a dictionary of commonly used terms in the domain, collecting information about the domain from various sources, defining the current, feasible and strategic domain for the company, and lastly performing a feasibility analysis.

After a term and its definition are entered (Figure 14), Holmes automatically parses all the data it stores and checks for occurrences of this term, which it transforms into hyperlinks. Such a hyperlink points to the definition of the term, which is opened in a separate window for easy referral (Figure 15). All textual data in Holmes is represented in HTML, thus, facilitating the browsing of information that is stored in different data entities. When editing textual data, the user of the system is given the opportunity to use a tool of his/her preference or the Holmes' build-in HTML editor. The configuration of a third-party HTML editor is straightforward and requires only the full path to an executable. In the provided example, HTML editing is performed with Microsoft Front Page (Figure 14).

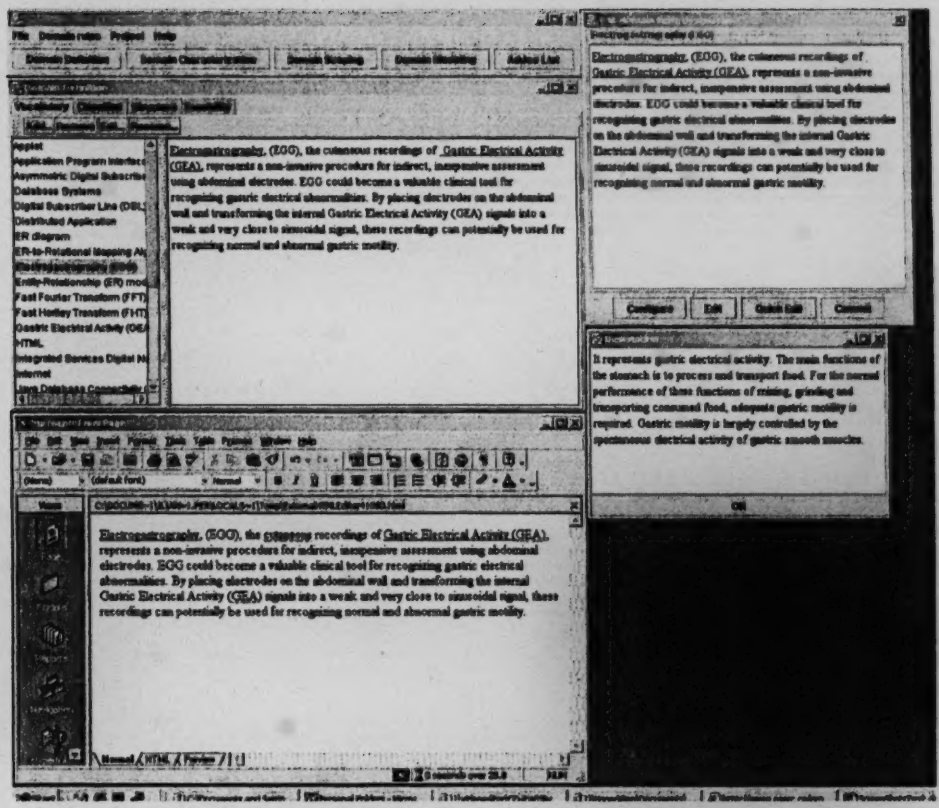


Figure 14: Domain vocabulary

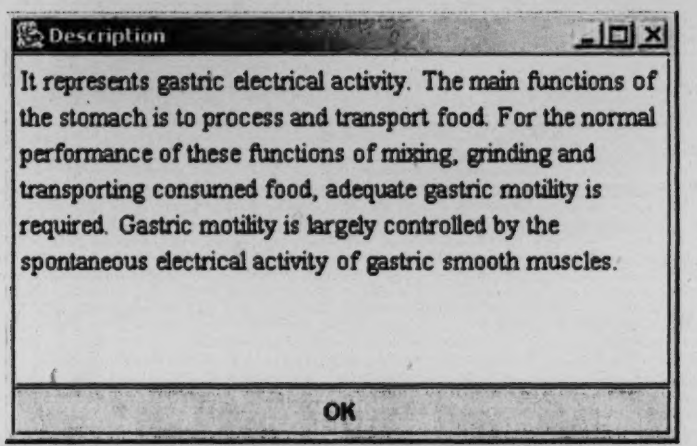

Description
It represents gastric electrical activity. The main functions of the stomach is to process and transport food. For the normal performance of these functions of mixing, grinding and transporting consumed food, adequate gastric motility is required. Gastric motility is largely controlled by the spontaneous electrical activity of gastric smooth muscles.
OK

Figure 15: Domain description

During the activities of collecting domain information (Figure 16) and performing feasibility analysis (Figure 17) the data entered is monitored for domain term occurrences. The reference model for domain terms and the fact that each term is unique,

non-duplicated entity (in reality an object) makes the propagation of any change to a term instant in all documents that include it.

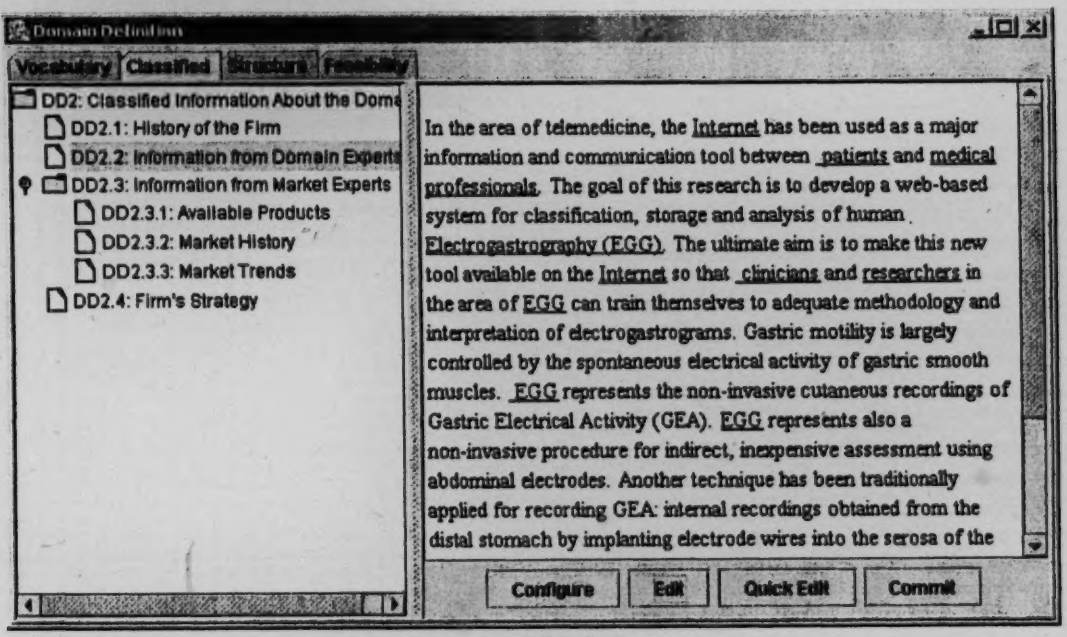

Vocabulary
Classified
Structure
Feasibility
DD2: Classified Information About the Dome
DD2.1: History of the Firm
DD2.2: Information from Domain Experts
DD2.3: Information from Market Experts
DD2.3.1: Available Products
DD2.3.2: Market History
DD2.3.3: Market Trends
DD2.4: Firm's Strategy
In the area of telemedicine, the Internet has been used as a major information and communication tool between patients and medical professionals. The goal of this research is to develop a web-based system for classification, storage and analysis of human Electrogastrography (EGG). The ultimate aim is to make this new tool available on the Internet so that clinicians and researchers in the area of EGG can train themselves to adequate methodology and interpretation of electrogastrograms. Gastric motility is largely controlled by the spontaneous electrical activity of gastric smooth muscles. EGG represents the non-invasive cutaneous recordings of Gastric Electrical Activity (GEA). EGG represents also a non-invasive procedure for indirect, inexpensive assessment using abdominal electrodes. Another technique has been traditionally applied for recording GEA: internal recordings obtained from the distal stomach by implanting electrode wires into the serosa of the
Configure
Edit
Quick Edit
Commit

Figure 16: Classified information

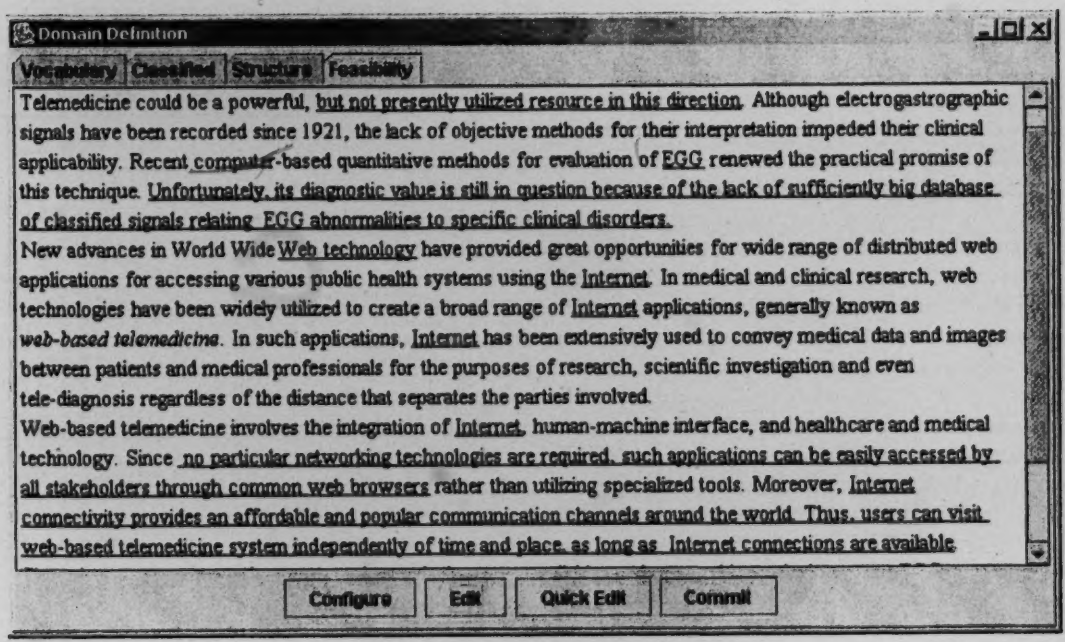

Domain Definition
Vocabulary
Classified
Structure
Feasibility
Telemedicine could be a powerful, but not presently utilized resource in this direction. Although electrogastrographic signals have been recorded since 1921, the lack of objective methods for their interpretation impeded their clinical applicability. Recent computer-based quantitative methods for evaluation of EGG renewed the practical promise of this technique. Unfortunately, its diagnostic value is still in question because of the lack of sufficiently big database of classified signals relating EGG abnormalities to specific clinical disorders.
New advances in World Wide Web technology have provided great opportunities for wide range of distributed web applications for accessing various public health systems using the Internet. In medical and clinical research, web technologies have been widely utilized to create a broad range of Internet applications, generally known as web-based telemedicine. In such applications, Internet has been extensively used to convey medical data and images between patients and medical professionals for the purposes of research, scientific investigation and even tele-diagnosis regardless of the distance that separates the parties involved.
Web-based telemedicine involves the integration of Internet, human-machine interface, and healthcare and medical technology. Since no particular networking technologies are required, such applications can be easily accessed by all stakeholders through common web browsers rather than utilizing specialized tools. Moreover, Internet connectivity provides an affordable and popular communication channels around the world. Thus, users can visit web-based telemedicine system independently of time and place, as long as Internet connections are available.
Configure
Edit
Quick Edit
Commit

Figure 17: Feasibility Analysis

During the phase of domain characterization existing products are analysed from the perspective of features, quality attributes, interoperability and user's expectations (Figure 18). That information is used to determine how the products under development may evolve and what is the minimum set of features that should be implemented. Holmes not only provides a media for storing this information but also handles most of the bookkeeping. All products that are listed in the tools Diagrams of Value, Installed Base (from domain characterization) and Classified Information (from domain definition) are automatically kept consistent, i.e., if a new product is added in any of these tools it is automatically added in the others as well. For these product's attributes that are not visible in the tool of creation, default values are assigned. Doing so substantially alleviates the task of keeping the information entities that are represented in these tools consistent.

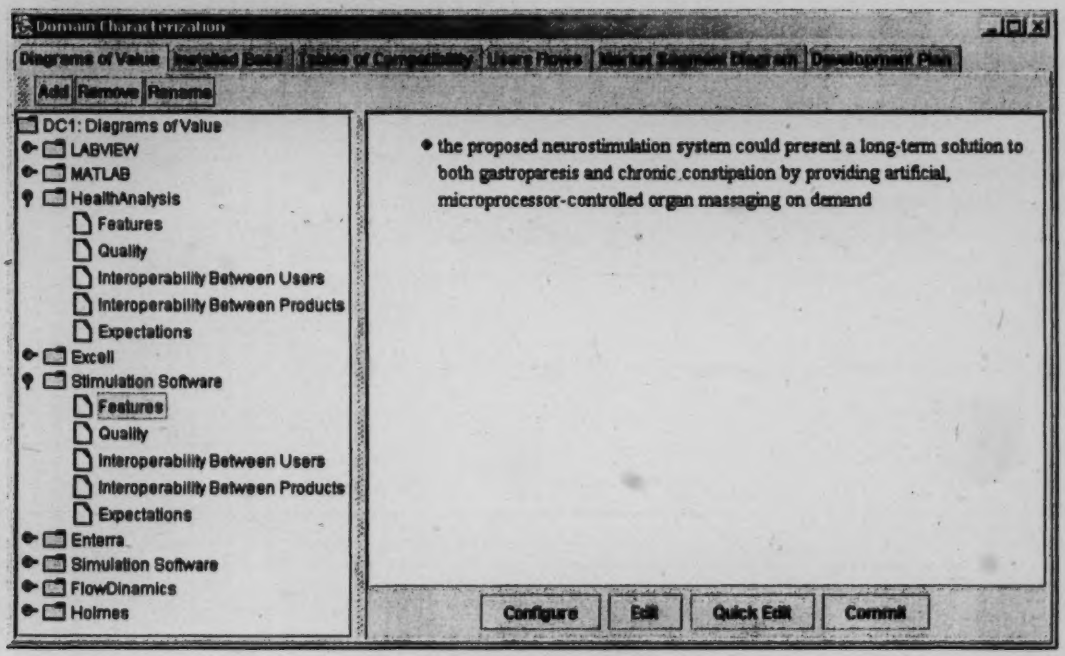


Figure 18: Diagrams of value

Holmes provides both a textual and a graphical representation for the compatibility types (Figure 19) and user flows (Figure 20) in the domain. In order to provide maximum convenience for experts with different background, Holmes enables all types of

modifications to be made either in the textual or graphical representation, while maintaining immediate change propagation between the two representations. Since the change propagation is maintained through the data queues (Section 5.2), it is possible for one expert to make modifications in the graphical representation, while other can remotely co-operate by working either on the graphical or textual representation.

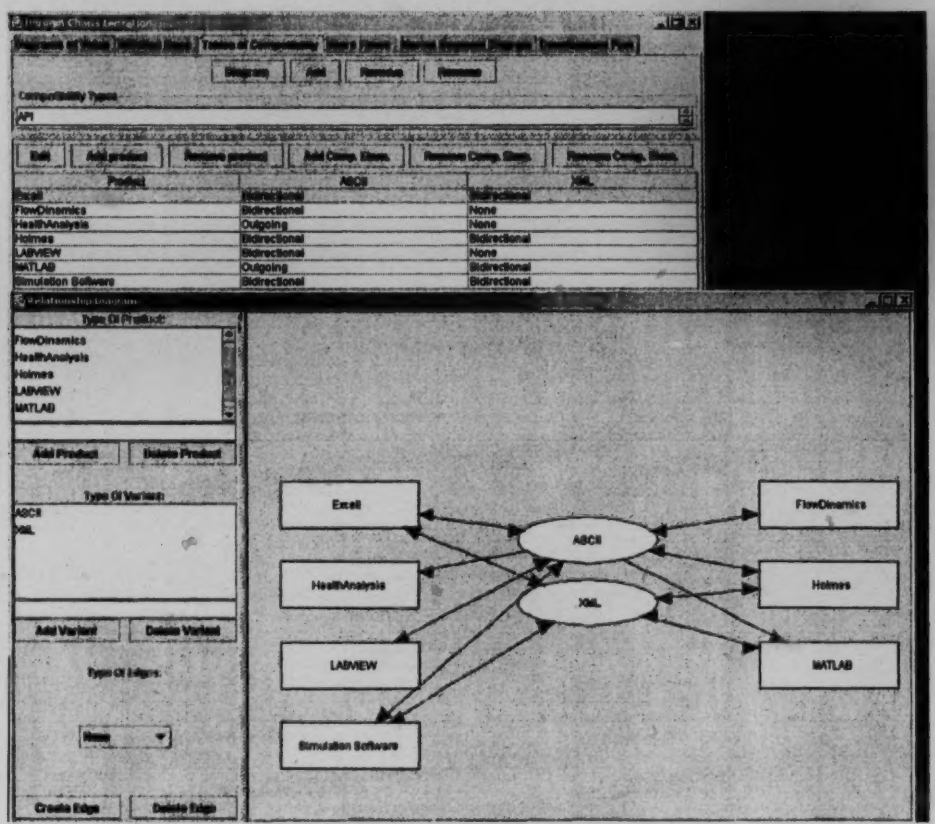


Figure 19: Compatibility types

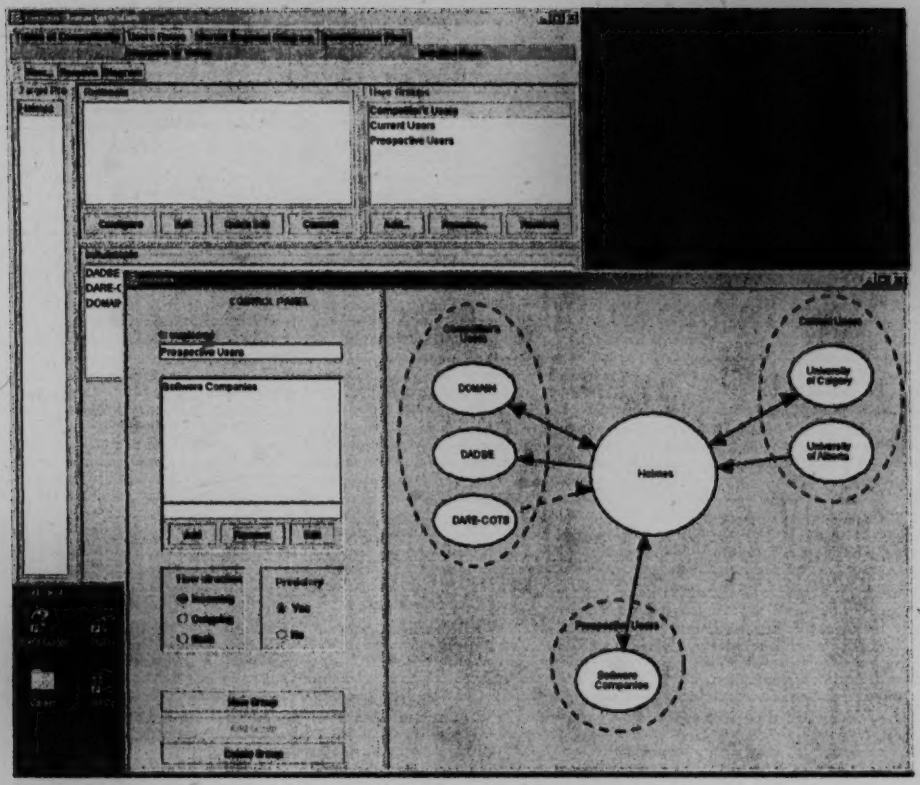


Figure 20: User Flows

The tool for Domain Scoping is used in commonality and variability analysis and provides the means for manipulating the outputs of that analysis: variation points, variants, and attributes. As described in section 3.3 a variation point is a conceptually common feature that can be implemented in different ways; the variations represent these different implementations. The variation points in a domain form a variability space populated with variants. The products in the domain are positioned on this variability space according to the variants they realize. Since there can be many variation points in a domain, which will result in a multidimensional space that is hard to visualize and reason about, projections of that multidimensional space are used. In the example provided on Figure 21 the projection of one side of the three-dimensional space is represented as three two-dimensional spaces.

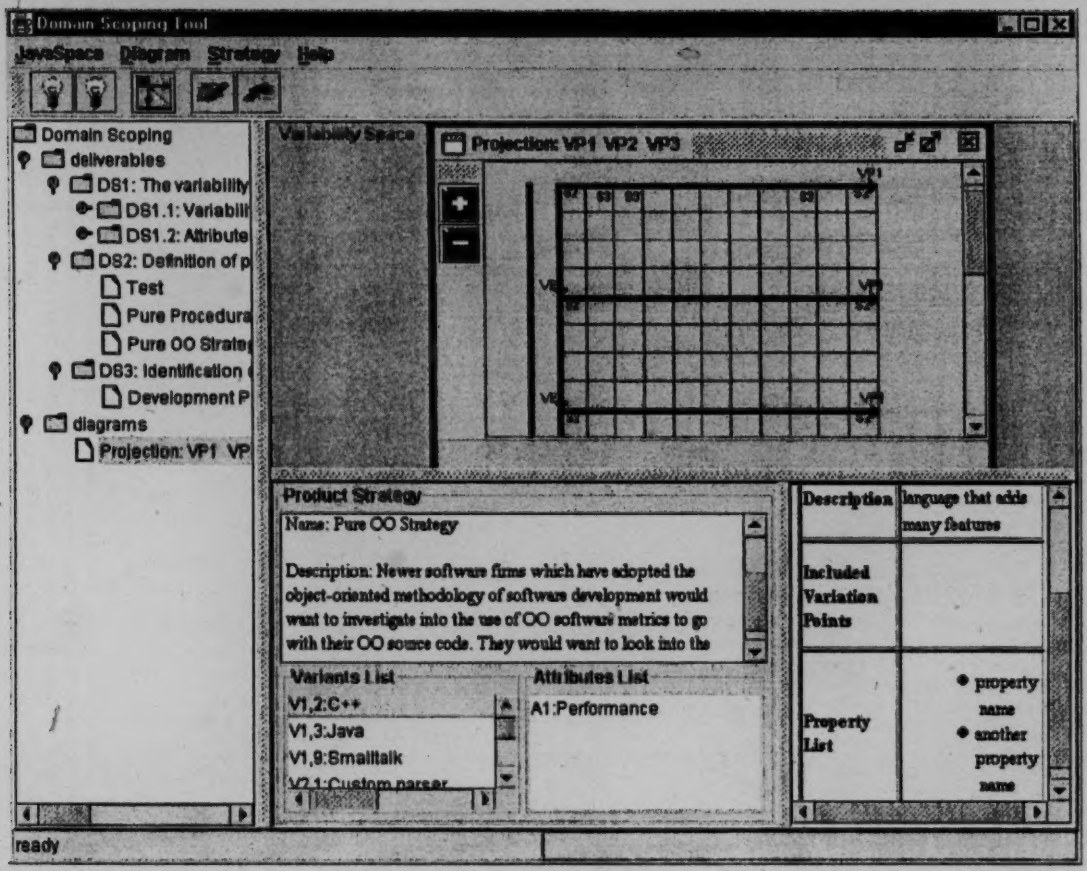


Figure 21: Domain Scoping

The tool for Domain Scoping automatically generates the diagrams for each projection presenting the variation points that form this projection, the variants that populate it and the products and strategies that are positioned on it (Figure 21). Thus, the substantial effort of manually creating and updating that otherwise helpful visual representation is waived.

The domain modelling tool deals with the creation and enhancement of domain and product models including use case models and analysis models. Since there is a number of existing third party tools, which provide sophisticated support for the process of business and software systems modelling, integration rather than in-house development was chosen. Holmes supports data and some limited control integration with Rational Rose model tool (Rational, 2002). The main reason for the limited success in achieving control integration was the poor quality of free Java-to-COM (Microsoft, 1995) bridges.

Holmes stores all models that are created by Rational Rose, so when a software practitioner needs to edit or review a given model he/she selects it and presses the "Edit" button (Figure 22). This will automatically invoke Rational Rose and open within it the selected model. The Rational Rose model files (*.mdl) are also parsed to extract Object-Oriented metrics (Chidamber and Kemerer, 1994), so that the models they represent can be analysed by the critiquing system.

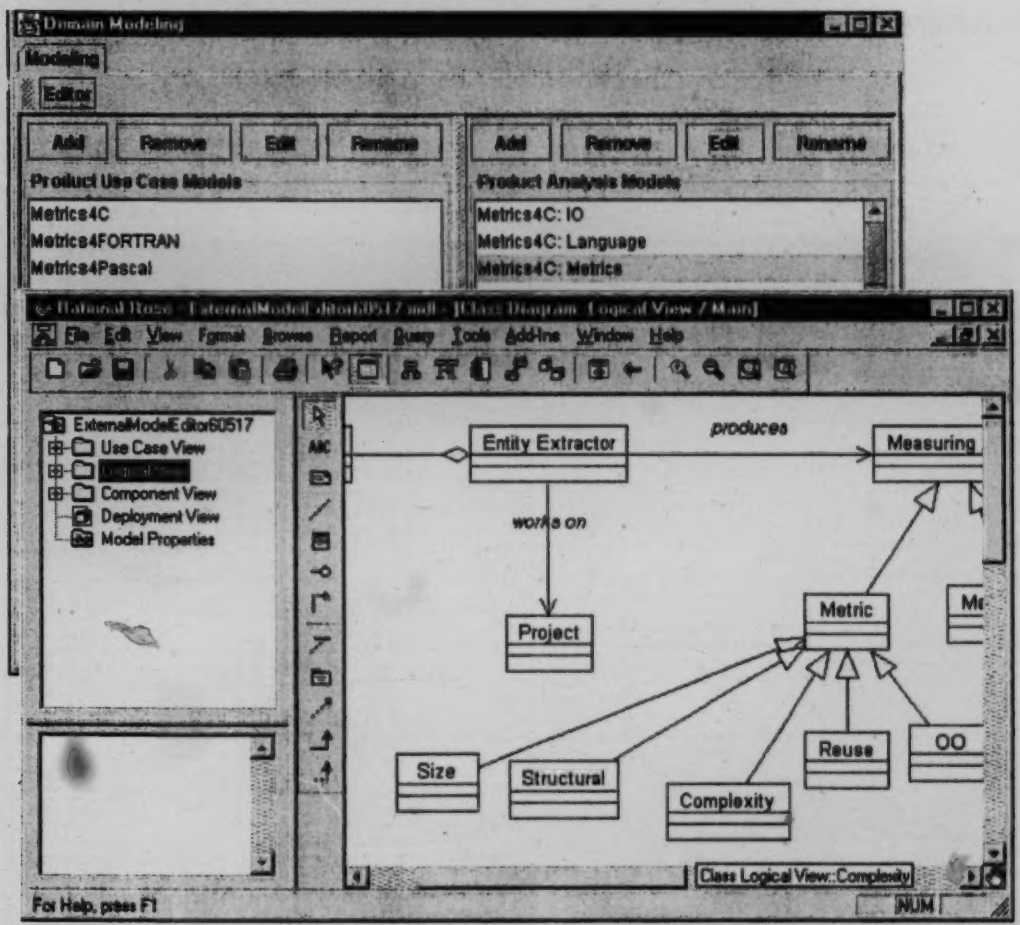


Figure 22: Domain Modelling

5.6 Summary

Holmes is a tool that supports the Sherlock methodology for domain-based SPL development. The proposed implementation of Holmes is based on a framework developed using Sun's implementation of a tuple space, called JavaSpaces. This framework provides for loose coupling among the integrated tools, and the framework. Since JavaSpaces are based on the object-oriented paradigm, any entity that is stored in the JavaSpaces should be an object. This condition, initially regarded as a constraint, enforced the development of a flexible, object-oriented mechanism for communication among the tools, as opposed to standard message passing. The mapping of Holmes to the recognized tool requirements is presented in **Table 4**.

Feature	Allows queries	Change consistency management	Link consistency management	Multi-user support	Tool integration support	Semantic support
JavaSpaces		✓		✓	✓	
Data repositories		✓			✓	
Event queue		✓				
Generic tool adaptors					✓	
Holmes markup language					✓	
Critiquing system			✓			✓
Hyperbolic tree view			✓			
Prolog scripting	✓					✓

☐ - Not/Poorly Supported ✓ - Supported

Table 4: Mapping to requirements

Future work will be focused on analysing the possibilities for uniform data storage. This should result in constructing a meta-language having the necessary descriptive power to represent relationships utilized by all incorporated tools.

Another component of future work will be focused on the further development of the Critiquing System, which will identify possible pitfalls in a proposed design and suggest

probable improvements. These suggestions will be based on existing reusable object-oriented design pattern.

The third component of future development will be the improvement of the visualization of domain information. The use of hyperbolic browser promises to give a better representation of existing data dependencies and relationships. This is closely related to the problem of traceability of information among the different phases of SPL development.

6 Case Study

In this section a case study of using Holmes to perform domain analysis in the field of software controlled Gastrointestinal Stimulation is presented. The use of software-controlled gastrointestinal (GI) stimulators to regulate gastric activity promises to revolutionize treatment of GI-related diseases. Because this software area is relatively young, it is difficult to propose a concrete and detailed realization of a product line in GI. However, a comprehensive analysis of the domain identified the main requirements and problems that should be considered when developing software systems in that domain. The analysis identified the critical areas for further investigation, including the necessary conditions for the implementability of a software-controlled GI stimulation system. It also offered an initial plan for product development providing incremental delivery of value.

6.1 Domain Definition

As described in Section 3, the purpose of the first phase in Sherlock methodology is to gather information about the domain from several sources including history of the organization, domain experts, market experts, etc. By compiling and systemizing the information Domain Definition reduces the risk of performing DAE in the wrong direction. More precisely, in this case Domain Definition identified the boundaries of the GI domain and gives a conclusion whether it is feasible to continue with DAE.

6.1.1 Information about the domain

Gastrointestinal electrical stimulation has been proposed as a treatment for a variety of gastrointestinal motility disorders ranging from abnormal gastric emptying and gastroparesis, to chronic idiopathic constipation (Everhard, 1994) (McCallum et al., 1998). In fact, a commercial system already exists that uses implanted electrodes to treat severe cases of gastroparesis, which has been FDA approved for humanitarian use only (Medtronic, 2000). However, existing pacing (McCallum et al., 1998) and entraining (Everhard, 1994) solutions failed to produce convincing evidence for invoked movement

of gastrointestinal content associated with the stimulation. In contrast, functional gastrointestinal stimulation is a treatment, which involves the use of sophisticated software-controlled microprocessor-based neurostimulator to restore impaired gastrointestinal motility utilizing sets of circumferentially implanted electrodes supplied with synchronized trains of bipolar high-frequency (over 50 Hz) voltage trains with variable amplitude and duration (Mintchev et al., 1998). Complex software modelling of gastrointestinal organs as electromechanical engineering systems preceded the design of a prototype of functional neurostimulator. That modelling was pivotal for the derivation of the optimal stimulation parameters utilized in the software development of the device itself (Mintchev et al., 1997).

In order to enable a successful transition from the current prototypical software to a complete software-controlled miniature GI simulation system, an analysis of the software domain is proposed to investigate the areas of risk, uncertainty, and any other factors that future development must successfully resolve.

The general domain of functional gastrointestinal stimulation can be split into two main aspects: simulation software for modelling artificially-invoked movement of gastrointestinal content, and embedded system software that controls gastrointestinal neurostimulators implanted in the human body. Consequently, the information gathered will be divided in the above-mentioned groups.

An embedded system will implement a method for stimulating the smooth muscles of gastrointestinal organs that was shown to be successful in a number of studies (Mintchev, et al. 1998). It uses a number of circumferentially placed electrodes to propagate electric stimulation. There are a few parameters of the electric stimulation that are of vital importance, in order to produce an optimal contraction of the smooth muscles:

- Length of the stimulating electrodes
- Scope of contractions and separation between the successive electrode sets
- Duration of the stimuli applied to the electrode sets
- Phase lag between the stimuli applied to sequential electrode sets
- Amplitude of the stimulating voltage

- Current sinking capabilities of the stimulating device

The specific values for the above-mentioned parameters, voltage and frequency of electrical current have been described for the stomach (Mintchev, et al. 1998) and for the colon (Rashev et al., B). It is expected that these values would differ for other gastrointestinal organs within a certain range. Flexibility in programming these parameters is highly desirable from the point of view of adjusting the stimulator to be patient specific as well. Therefore, the software of the embedded system should be customizable with respect to these parameters.

The simulation software should provide means for visualizing and verifying the effects of a particular electrical stimulation. In order to do so it should be based on theoretical model of mechanically active GI organs and more specifically the tissues that compose them. There are currently two major approaches to modelling gastric electrical activity (Rashev et al., 2000):

- Passive models, employing networks of coupled relaxation oscillators, which represent the spontaneous neural stimulation of the excitable smooth muscle tissue of a particular organ by mapping the expected potential or current distributions generated by the local nervous system.
- Active models, utilizing the dipole theory to describe vector characteristics of the electric field produced by the excitable tissues of an organ, or representing the changes in the magnetic field caused by alternations of the GEA.

The passive and active models provide precise representation of the spontaneous electrophysiological phenomena but do not cover in a detailed manner the electromechanical behaviour of the tissue. Thus, they are ill suited for simulating artificially stimulated contractions of tissues with electrical activity. Because of that the simulation software should be based on a novel object oriented modelling approach in electromechanical modelling of GI tissues. This model incorporates knowledge of the anatomy, electrophysiology, and mechanics of externally stimulated GI tissues and employs the stimulus-response principle in modelling them (Rashev et al., 2000). To address these issues the mechanical phenomena of GI organ contractions must be linked to the electrical phenomena that occur during GI organ contractions.

There are four dimensions to this problem domain:

- Electrical
- Mechanical – fluid and solid matter flow dynamics
- Medical/Physiological – refers to rejection, pain, long-term implantability, and other surgical issues
- Anatomic – the shape of the GI organ being modelled

Only the electrical and anatomic dimensions are considered for the proposed software system. The mechanical dimension is not considered because it is believed to be more efficient to exploit existing flow dynamics software rather than develop custom mechanical modelling tools from scratch. Medical/physiological dimensions are outside the scope of the software system, since they are related to prolonged testing on chronic animal models.

6.1.2 Domain Vocabulary

Since the problem domain is strongly related to medicine most of the terms come from that field. Consequently, it will not be practical to enumerate them and their explanation in the current document. Detailed vocabulary can be found in previously published articles on the subject in major medical journals.

6.1.3 Information about the market

The Medtronic Enterra therapy system (Medtronic Enterra Therapy, 2000) is a commercial product that targets the same need as the proposed system. However, unlike the proposed neurostimulator, Enterra does not use simulation and analysis to control electrode use and placement, and employs subHertz stimulating frequencies, thus providing entraining of spontaneously existing gastric electrical activity, rather than neuroelectrical stimulation resulting in circumferential contractions. The existence of Enterra does suggest that there is a viable market for software-controlled GI stimulation. Detailed statistics (for the United States) on the two major disorders that can be treated with the proposed neurostimulation technique, severe gastroparesis and chronic constipation, is provided by the National Institutes of Health (Everhard, J.E., 1994). It can

be estimated that several million people in North America suffer from chronic constipation, while at least 20% of patients with Type 1 Diabetes develop gastroparesis. Increasing number of cases with idiopathic gastroparesis has been reported as well.

6.1.4 Overall strategy

The overall strategy of the current development effort is as follows:

- Determine whether the approach of using simulation is of practical value by providing a framework for off-line simulation and analysis;
- Provide ways for utilizing the developed parametric electromechanical model for displaying more sophisticated animation and visualization;
- Develop a real-time system that interfaces with the embedded control of sequentially implanted sets of stimulating electrodes and thus provide data, which can be further processed;
- Synchronize the work of the model with the actual neurostimulation, so that patient-specific configurations of the stimulation parameters can be quickly and efficiently obtained during the operation for the electrode implantation.

Given these goals, there are two general areas of interest: GI simulation software and GI embedded system software.

6.1.5 Definition of domains

Sherlock defines several domain boundaries:

- *Current domain*: Domain of currently existing and under development software systems.
- *Strategic domain*: Domain that the firm is most interested in and would like to expand to in the near future.
- *Feasible domain*: Domain that encompasses all long-term possibilities for future expansion.

Table 5 shows the domains for the two general areas of interest:

Domain	GI Simulation Software	GI Embedded System Software
Current	Stomach and colon	GI simulation software
Strategic	All GI organs	Software controlled GI stimulation software (includes simulation and embedded) system
Feasible	All mechanically active human organs	General software-controlled implantable neurostimulator

Table 5: Domains

6.1.6 Feasibility analysis

The GI simulation (GIS) software is envisioned as a modular, expandable system. The development of the current prototype confirms the feasibility of building a software GI stimulation simulator. The market analysis shows that at present there is no tool to offer similar functionality. In addition such GIS simulator will be valuable for the future implantable neurostimulators by providing a way to carry out an adjustment of the GI stimulators before they are permanently implanted in a human body. Concerning the GI stimulation embedded system, its feasibility and value have been already supported by the currently existing commercial treatment, the available digestive disease statistics, and the extensive research carried out in that field.

6.2 Domain Characterization

Domain Characterization gathered the requirements in the domain. It provided precise synopsis of what market needs the domain was meant to address. These needs were expressed by using the terms commonly used in the analysed domain. Domain Characterization also gave a description of the internal and external value of the products.

6.2.1 Simulation software

The internal value of the simulation software is represented by the possibility to simulate the results of GI treatment and adjust the positioning of the electrodes and the parameters of stimulation before the actual surgical placement of an implantable stimulation system

is performed. This will decrease the possibility of any adverse events caused by wrong parameterization of the stimulator and will yield best results from the very beginning of the treatment.

The key features of the proposed GI simulation software are equivalent to the key features of a GI electro-mechanical model:

- The ability to modify stimulation parameters and the scope of the artificially-produced contractions;
- Visualization of the modelled mechanical contractions of the GI organ due to the applied virtual stimuli;
- Accurate reflection of the scope and duration of contractions with respect to the scope and duration of the applied virtual stimuli;
- The ability to manipulate the virtual stimuli in order to achieve virtual propagated contractions.

The accuracy and precision of the visualization is particularly important because medical practitioners will be assessing the correctness of the propagation of contractions based solely on the visualization.

As mentioned in the Domain Definition section, the proposed system will interface with a flow dynamics software package to handle more sophisticated mechanical issues. Ideally the simulation software would incorporate the flow dynamics package so that any particular simulation (*e.g.*, the gastric emptying of a solid meal with a particular viscosity) would be visually presented in real time simultaneously with the stimulating pattern. Thus, the resulting presentation would visualise the real-time response of a stimulated organ. A critical future decision will be concerned with which particular flow dynamics package will be used and the corresponding method of interfacing, whether through files or a programming interface.

The external value will highly depend on the existence of other medical software systems in related fields.

6.2.2 Embedded software

The internal value of the embedded software system is represented by the effects caused by GI stimulation when applied in cases of severe gastroparesis, chronic constipation, etc. While the existing Medtronic Enterra Therapy claims significant reduction of the symptoms of nausea and vomiting in gastroparesis when other pharmacological treatments are ineffective, the proposed neurostimulation system could present a long-term solution to both gastroparesis and chronic constipation by providing artificial, microprocessor-controlled organ massaging on demand.

Another aspect of the internal value of the embedded system is related to the prevention of possible adverse events when GI stimulation is applied. Special care should be taken in analysing the components used in the design of the actual stimulating system so that electrode problems, device infections, device erosion, device migration, etc. are prevented. Given the potential problems related to electrode movement, it seems appropriate to consider whether features related to electrode tracking and software handling of overall system failures should be considered within the overall scope of the product line.

The external value of the embedded system is considerably limited by the nature of this device. Further discussion of that aspect at the current level of development will be ungrounded and superficial.

6.2.3 Market segments and users

The users or the market segment that a product is targeting have major influence on feature selection decisions. The ultimate and primary market segments for this effort are obviously patients requiring treatment for a GI-related motility disorders and the doctors treating these patients. However, it is also useful to examine secondary and tertiary user groups, especially those that can support interim software products leading to the eventual development of the embedded system. Some possible non-primary market segments for the simulation software include students and researchers. A system for

students might have to provide more assistance than a system designed for experts would require, while a system for researchers may have to provide more features for experimentation.

In general, taking advantage of interim market segments is a risk-mitigation strategy. Usability of the simulation aspects of the system can be tested with real users independent of other considerations that an embedded system would introduce. Even if the embedded software system can be completed quickly, there are other obstacles of practical importance that may prevent the overall hardware-software system, from being put into production or even integrated. One such issue is the availability of power sources that have suitable size and duration.

6.3 Domain Scoping

Domain Scoping is the phase in which variability analysis for the domain is carried out. The existence of a number of products in a domain means that they share some commonalities. These commonalities roughly show what the products actually do. The specifics of products are denoted by the differences among them. In the case of a single product Domain Scoping is used to point out the possible features the product can implement and suggest a number of strategies for its development.

6.3.1 Variation points and attributes

Variation points, also known as hot spots, are aspects in a software product line that will be addressed differently by different products. In other words, a variation point is a feature of commonalities that is implemented differently in different products. For example, Operating System could be a variation point with the variants, or particular instantiations of a variation point, being Windows, UNIX, and Real Time Operating System (RTOS). Variants may or may not be mutually exclusive depending on the variation point.

Simulation Software

Variation Point 1: Organs that can be simulated

- **Variant 1,1:** Esophagus
- **Variant 1,2:** Stomach
- **Variant 1,3:** Colon

Variation Point 2: Content that can be simulated

- **Variant 2,1:** None
- **Variant 2,2:** Fluids
- **Variant 2,3:** Solids

Variation Point 3: Output formats

- **Variant 3,1:** Plain text
- **Variant 3,2:** XML
- **Variant 3,3:** Binary

Variation Point 4: Operating system

- **Variant 4,1:** DOS
- **Variant 4,2:** Windows 95/98
- **Variant 4,3:** Windows NT/2000
- **Variant 4,4:** UNIX
- **Variant 4,5:** RTOS

Attributes are similar to variation points except that they deal with non-functional aspects and thus tend to have more widespread implications to design and implementation decisions

Attribute 1: Performance

- **Constraint 1,1:** Response time in the order of milliseconds
- **Constraint 1,2:** Response time under 5 ms
- **Constraint 1,3:** Response time in the order of microseconds

Attribute 2: Reliability

- **Constraint 2,1:** Mean Time to Failure (MTTF) in the order of hours
- **Constraint 2,2:** MTTF in the order of months
- **Constraint 2,3:** MTTF in the order of 50 years

Attribute 3: Usability

- **Constraint 3,1:** An average student can learn the system in at most a few hours
- **Constraint 3,2:** A domain expert can learn the system in at most a few hours
- **Constraint 3,3:** A domain expert requires at most 3 months training to learn the system

Embedded System

Variation Point 1: Organs that can be simulated

- Variant 1,1: Esophagus
- Variant 1,2: Stomach
- Variant 1,3: Colon

6.3.2 Product strategies

A product strategy is essentially a set of choices made on which variants and which attribute constraints are appropriate for a targeted market segment. Strategies are also not necessarily mutually exclusive and can in fact be combined and implemented simultaneously if desired. Several proposed strategies follow:

Strategy 1: Training tool

This strategy would target students and instructors. Although it would be advantageous to be able to simulate more organs and show movement of content, it would likely be sufficient to simulate a major GI organ like the colon in order to communicate concepts. Output formats can be of any form. The operating system will depend on the particular situation but will unlikely be a RTOS. Performance can be soft real-time in the order of milliseconds or even worse. Reliability can be in the order of hours, which is probably longer than any lecture or lab. Usability on the other hand, has to be very good so that students and instructors are not frustrated by the system.

Strategy 2: Clinical research tool

Clinical researchers would likely require a system that could simulate more GI organs and the movement of content within them. In order to improve the ability to share information with other researchers, a XML output format would be preferred. Typical university research computers use either UNIX or Windows NT/2000 although one should pay attention to exceptions. The performance of the system should be as close to hard real-time as possible though a response time under 5 ms is probably adequate. Reliability on the order of months should be sufficient. Usability should be tailored to domain experts and should be learnable in at most a few hours.

Strategy 3: Embedded system plus support

The support system for an embedded strategy would require the ability to simulate the organ where the electrodes would be installed. It is unlikely that all organs would be attempted simultaneously. Probably the first organ that would be attempted would be the colon. Improved content simulation may also be a feature gradually added. This will depend on the existence of empirical evidence from implant patients showing that it will improve the system sufficiently in terms of quality-of-life. Sharing of information is mostly between the simulation and embedded control components suggesting that a binary format is more suitable for efficiency. The embedded control component would definitely need to run on a RTOS but the simulation component could run on a less strict

operating system. A common consumer OS would be ideal for ease of use, however, the real-time considerations suggests that UNIX or Windows NT/2000 might be more realistic. To reduce cost, an open source UNIX like a Linux or BSD distribution would be ideal.

6.4 Domain Modelling

With the current prototype software, the sequence and duration of activities is as follows:
Scale organ and adjust parameters to be case-specific - ~1 day
Specify electrode placement, simulate static contraction and check amplitude of stimulus, time constants of contraction and relaxation, and electrode positioning to get required lumen occlusion - ~2 days
Simulate and examine results, manually adjust parameters to obtain synchronization patterns needed to move stimulus in a multi-electrode setup- ~5 days.
The total duration of this semi-automatic process is a little over a week.

6.4.1 Scenarios

There are several scenarios that proposed systems would address, including some extracted from the current activity sequence.

- *Scale organ*: The user adjusts the geometric parameters of a generic organ to match the current specific case.
- *Simulate static contractions*: The user specifies the placement of electrodes on the model. The system shows the amplitude and time characteristics of the stimulus, the electrode positioning and the resulting lumen occlusion in the model.
- *Simulate dynamic contractions:* The user adjusts parameters to get timing synchronization of the stimuli produced by multiple electrode sets needed to propagate the lumen-occluding contractions.
- *Simulate contents:* The user specifies the type of content and its viscosity. The system calculates the effect on volume displacement and shows the resulting movement of fluids or solids within the organ model.

- *System level simulation:* Individual organ simulations are combined into a unified system-level simulation.

6.4.2 Analysis Model

The analysis model (Figure 23) acts as a structured concept map that abstracts out the less important details to describe the overall system in simpler manner. The sequence diagram in Figure 24 describes the dynamic behaviour of the system. An ElectrodePair will stimulate an OrganTissue. OrganTissues are connected through OrganJunctions, which are themselves a type of OrganTissue. Content is moved through the overall GI system from the responses of the OrganTissue to ElectrodePair stimulation. Meanwhile, the WorldView is observing all this behaviour and displaying it to the user.

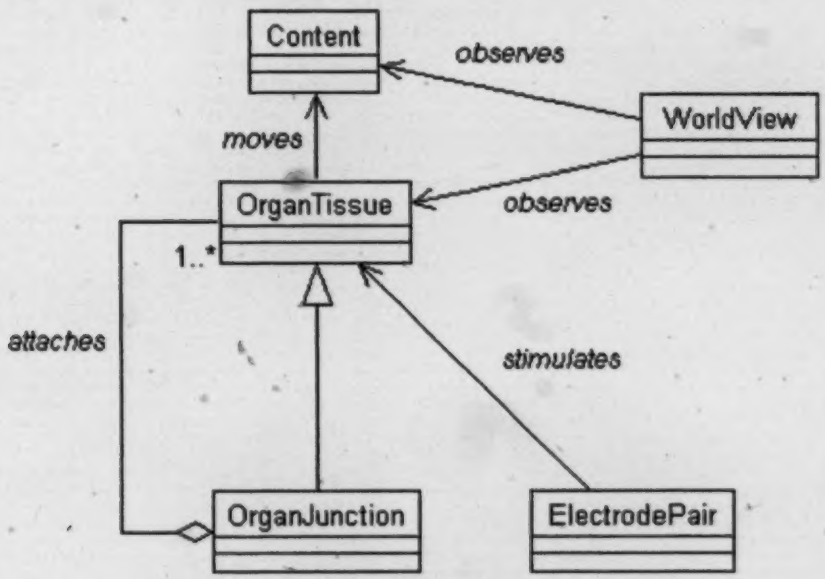


Figure 23: Analysis Model

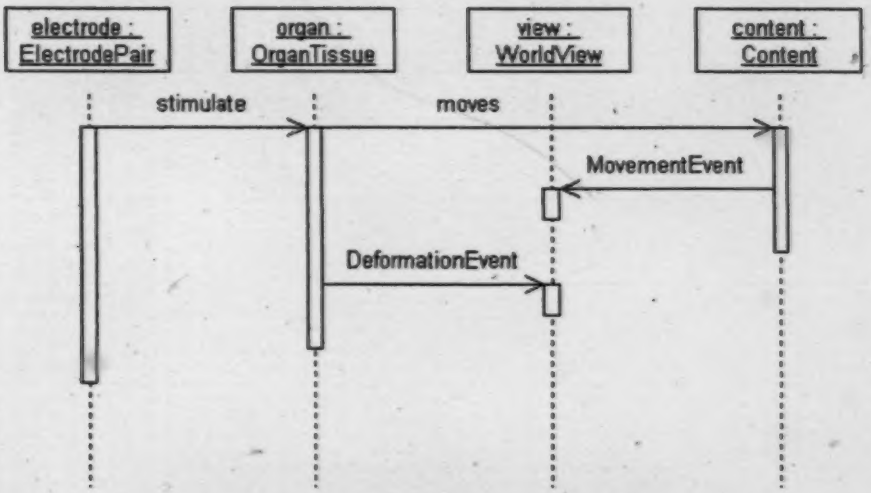


Figure 24: Sequence Diagram for Analysis Model

Table 6 lists the classes, example instances, and their corresponding responsibilities.

Class	Example Instances	Responsibilities
OrganTissue	Stomach, ColonicSmoothTissue	Knows shape Calculates deformation of its shape in response to a stimulus
OrganJunction extends OrganTissue	EsophagealJunction, IlealJunction	Knows shape Calculates deformation of its shape in response to a stimulus Attaches two OrganTissues
ElectrodePair	ElectrodePair	Stimulates an OrganTissue Knows its location on OrganTissue Calculates 2D mapping of current density distribution
Content	FluidContent, SolidContent	Knows shape and viscosity Calculates displacement in response to deformation of containing OrganTissue
WorldView	WorldView	Tracks OrganTissue and Content objects and their changes Displays OrganTissue and Content objects according to their shape

Table 6: Classes and Responsibilities

6.4.3 Design Model

The current design model is shown in Figure 25.

There are several indicators of possible problems with the current design (Riel, 1999):

- Inheritance relationships that do not provide substitutability (Liskov, 1988)
- Dependencies between TissueExcitation and Tissue, ElectricallyStimulated and MechanicallyStimulated
- Is StructuralGITissueModel a "god" class?

Tissue doesn't "use" a Sheath; it "has" a Sheath. TissueExcitation doesn't "use" a Tissue; a Tissue is sent an Excitation (from either an electrode or content moving through it) and responds by deforming. Deformation might not have a logical dependency on the source of the Excitation that causes it. The class who is responsible for calculating the Deformation, which may not be the Deformation class itself, would need to know the source of the Excitation. The previously proposed analysis model assumed the Tissue would calculate the Deformation but design details may require to move this responsibility elsewhere.

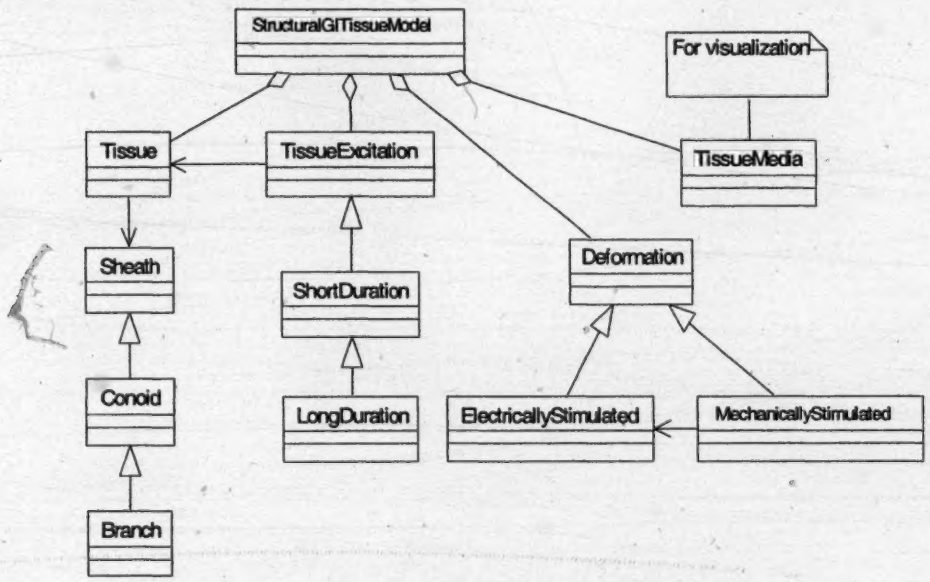


Figure 25: Current Design Model (Rashev et al., 2000)

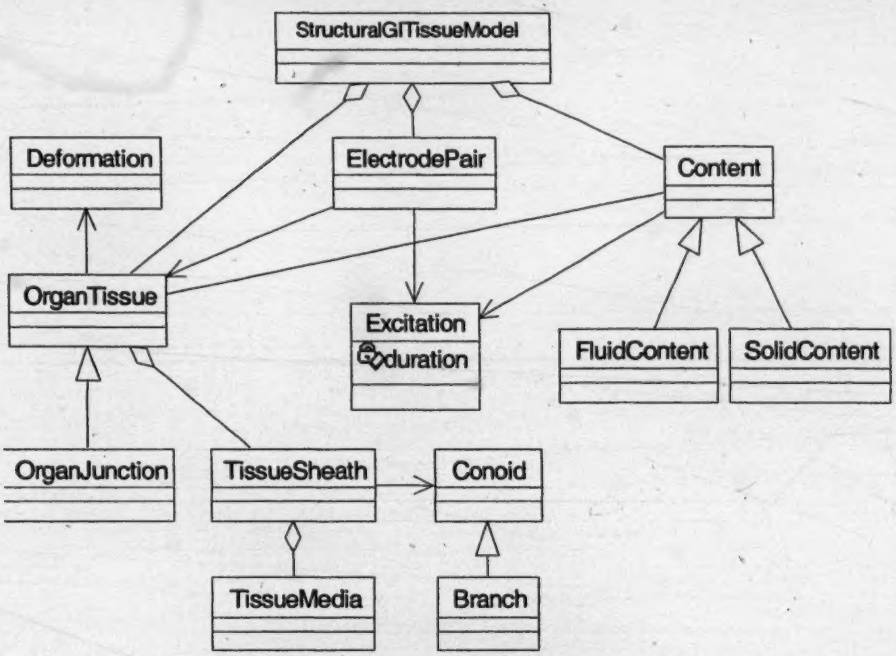


Figure 26: Proposed Design Model

A Tissue may react differently based on the duration of an Excitation but duration should simply be an attribute of an Excitation, not a sub-class. A Conoid is not a Sheath but is a specialization of the graphical object used to model a TissueSheath. The TissueSheath domain object should be kept separate from its graphical object representation. Figure 26 shows a proposed improved design model.

6.5 Summary

This case study presented the first domain analysis of gastrointestinal stimulation using Holmes support tool, developed as a part of this thesis. The goal was to identify the essential requirements, potential risks, and possible problems that can be encountered when developing software systems in the field of GI stimulation. One of the essential conclusions that were drawn was that the initial domain should be further split into software systems for GI stimulation simulation and software systems for actual GI stimulation. This division is intrinsic, since the former can be developed assuming relative abundance of computational resources and CASE tools, which can be exploited.

Conversely, the latter should be built within the resource-constrained environment of an embedded system. Furthermore, there are additional considerations that narrow the choice of hardware devices that can be used in a stimulation system. These considerations include, but are not limited to power consumption, size, implantability, medical suitability, etc.

An analysis and design model of a simulation system was proposed. By no means, the current design should be considered final. Further changes and improvements will be necessary as the software systems evolve.

The completed analysis aimed at facilitating the future development of software systems in that field and serving as a risk-mitigating factor. Particular attention was devoted on the boundaries of the domain and the products division within it.

7 Conclusion and Future Work

The strong competition in the software industry today necessitates substantial changes in the way software systems are developed. Each company faces the pressure for major improvement in software quality, on one side and reduced time-to-market, and lower development costs, on the other. The concept of Software Product Lines (SPL) development seems to be a viable solution in satisfying these counteracting requirements by embodying strategic, planned reuse, based on carefully analysed relation among the members of the product line.

The existing methodologies for SPL vary in the aspect of software development they identify as the most crucial for the successful adoption of SPL. While some approaches emphasise on modifying a company's structure and process so that separate organizational units deal with a different SPL practice, others put the stress mainly on changing the software process to one that will promote large-scale reuse, without explicitly defining the company's structure. Nevertheless, all existing methodologies demand building a software process and often an organizational structure anew (Chapter 2). Without doubt, if successful, such overhaul may provide major gains in the software development efficiency. However, a company's organization comprising a few units that must not have overlapping responsibilities and must not exchange personnel can be hardly viable in a small- and sometimes medium- sized company. Likewise, imposing a new and unfamiliar software process can initially lead to considerable frustration and delays in the achievement of current business goals. In brief, such major changes can be carried out successfully only in a fairly mature software development organization that is ready to make a significant upfront investment.

Another, more "lightweight", approach to adopting SPL development, systemized in a methodology called Sherlock (Chapter 3), is by defining the practices that have to be followed, leaving the company's structure loosely defined and augmenting with these practices the software process in place. In order to make the process of SPL development easier and more efficient, a design and implementation of a software environment that

supports Sherlock is proposed in the thesis (Chapter 5). This environment is designed to address issues that are poorly supported by existing tools, such as little support for the early phases of domain engineering, inadequate change propagation, multi-user and semantic support, etc. (Chapter 4) To verify the viability of Holmes, the support tool was applied to a case study in the domain of Gastro-intestinal stimulation. The study evidenced the advantages of using automated support as opposed to paper-and-pencil approach and revealed further improvements to the tool that will be beneficial.

In brief, the contribution of this thesis can be summarized in the following points:

- Identification of the essential requirements for an SPL support tool that are independent of the methodology that it will support
- Analysis of possibilities and choice of an architectural style that sufficiently matches the specifics of the problem addressed by an SPL support tool, i.e., SPL development
- Leverage of novel Object-Oriented technologies, such as Jini (Sun Microsystems Inc., 2001) and JavaSpaces (Freeman, et al., 1999) in order to address issues poorly supported by other existing tools
- Introduction of a mechanism for an easy adoption of SPL through the usage of a lightweight methodology and a support tool, which will shorten the required training of personnel and decrease the amount of initial investment
- Application of the developed tool to domain analysis in the field of Gastro-intestinal stimulation. This analysis identifies the essential requirements, potential risks, and possible problems that can be encountered when developing software systems in the field of GI stimulation.

Even though Holmes is a fully functional prototype, there are some areas in which it can be extended. In order to extend the link consistency management, a meta-language representing the dependencies among the data items pertaining to different phases of domain engineering should be developed. The critiquing system can be further enhanced by qualitative analysis in addition to the existing analysis based on quantitative measures Second, the possibility for implementing semi-automatic and automatic corrective actions

for some of the phases of domain engineering should be explored. Lastly, the browsing and navigating of domain information can be improved by introducing a hyberbolic tree viewer.

8 References

Arango G. (1994). Domain analysis methods. In Software Reusability, Ellis Horwood

Ardis, M. and Weiss, D. (1997). Defining Families: The Commonality Analysis. Proceedings of the Nineteenth International Conference on Software Engineering, IEEE Computer Society Press, May

Batory, D., G. Chen, E. Robertson, and T. Wang (2000) "Design Wizards and Visual Programming Environments for GenVoca Generators", to appear in IEEE Transactions on Software Engineering, URL: ftp://ftp.cs.utexas.edu/pub/predator/ieee-tse-99.ps

Bayer J., Flege O., Knauber P., Laqua R., Muthig D., Schmid K., Widen T., De Baud J. (1999). PuLSE: A Methodology to develop Software Product Lines, Symposium on Software Reusability

Bayer, J., D. Muthig and T. Widen (1999A) "Support for Domain and Variant Engineering: DIVERSITY/CDA" submitted to Automated Software Engineering '99

Braga, R., C. Werner, and M. Mattoso (1999) "Odyssey: A Reuse Environment based on Domain Models", Proceedings of the 1999 IEEE Symposium on Application-Specific Systems and Software Engineering & Technology

Booch G., Jacobson I., and Rumbaugh J. (1998). The Unified Modeling Language User Guide, Addison-Wesley

Chidamber, S.R. and C.F. Kemerer (1994). "A Metrics Suite for Object Oriented Design," IEEE Transactions on Software Engineering, 20(6).

Corkill, D. (1991). Blackboard systems. AI Expert 6(9):40-47, September

Dushin, F. (2000), JPL, http://sourceforge.net/projects/jpl/

Everhard, J.E. (1994). "Digestive Diseases in the United States: Epidemiology and Impact. NIH Publication No. 94-1447, US Department of Health and Human Services, National Institutes of Health, National Institute of Diabetes and Digestive and Kidney Diseases, Washington, DC.

Frakes, W., R. Prieto-Diaz and C. Fox (1998) "DARE: Domain analysis and reuse environment", Annals of Software Engineering, 5(1998)

Fisher G., Lemke A., Mastgalio T., and Morch A. (1991). The Role of Critiquing in Cooperative Problem Solving. ACM Transactions on Information Systems, Vol. 9, No. 3, April

Fisher G., Girgensohn A., Nakakoji K., and Remiles D. (1992). Supporting Software Designers with Integrated Domian Oriented Design Environments. IEEE Transactions on Software Engineering, vol. 18, no. 6, June

Fisher G., Nakalkoji K., Ostwald J., Stahl G., and Sumner T. (1993). Embedding Critics in Design Environments. The Knowledge Engineering Review Journal, Special Issue on Expert Critiquing vol. 8, no. 4

Freeman E., Hupfer S., and Arnold K. (1999). "JavaSpaces Principles, Patterns, and Practice" Addison-Wesley

Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

Gelernter, D., "Generative communication in Linda", ACM Transactions on Programming Languages and Systems, 7(1), 1985

Gotel, O., and A. Finkelstein. (1994) "An analysis of the requirements traceability problem", Proceedings of the 1994 International Conference on Requirements Engineering

Griss, M., J. Favaro, and M. d'Alessandro (1998) "Integrating Feature Modeling with the RSEB", Proceedings of the Fifth International Conference on Software Reuse

Jacobson I., Griss M., Jonsson P. (1997). Software Reuse – Architecture, Process and Organization for Business Success, Addison-Wesley

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., and Irwin J., (1997). Aspect-Oriented Programming. Proceedings of the European Conference on Object-Oriented Programming

Kruchten, P., (1995). Architectural Blueprints – The "4+1" View Model on Software Architecture. IEEE Software 12 (6), November.

Kang, K., et al. (1990) *Feature-Oriented Domain Analysis (FODA) Feasibility Study* (CMU/SEI-90-TR-21, ADA 235785). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University

Liskov, B., (1988). Data Abstraction and Hierarchy, SIGPLAN Notices, 23(5).

Loral Defense Systems (1996) "User Manual: ELPA Domain Generation Environment (EDGE) Version 2.0", Technical Report STARS-PA 19-S001/002/00

McCallum, R., J. Chen, Z. Lin, B. Schirmer, R. Williams, and R. Ross, (1998) "Gastric pacing improves emptying and symptoms in patients with gastroparesis", Gastroenterology, 114: 456-461.

Medtronic Enterra Therapy, (2000). URL: http://www.medtronic.com/neuro/enterra/; October 17

Microsoft (1995). The COM Specification. http://www.microsoft.com/com/resources/comdocs.asp

Mintchev, M. P., and K.L. Bowes, (1997) "Computer model of gastric electrical stimulation", Ann. Biomed. Eng., vol. 25, pp. 726-730.

Mintchev, M.P., C. P. Sanmiguel, S. J. Otto and K. L. Bowes, (1998). "Microprocessor controlled movement of liquid gastric content using sequential neural electrical stimulation", Gut, 43:607-611.

Mollaghasemi M. and Pet-Edwards J., (1997). Making Multiple-Objective Decisions. IEEE Computer Society

Parnas D.L., (1972). On the Criteria To Be Used in Decomposing Systems into Modules, Communications of the ACM, December

Prieto-Diaz, R., (1991) "Implementing Faceted Classification for Software Reuse," Communications of the ACM, Vol. 35, No. 5.

Predonzani, P., G. Succi, and T. Vernazza (2000) A Domain Oriented Approach to Software Production, Artech House Publisher Inc.

Prosperity Heights Software (2000) "Metaprogramming Text Processor", http://www.domain-specific.com/MTP/index.html

Rashev, P., M. P. Mintchev, and K.L. Bowes, (2000). "Application of Object-Oriented Programming Paradigm in Three-Dimensional Computer Modeling of Mechanically Active Gastrointestinal Tissues", IEEE Transactions on Information Technology in Biomedicine, vol. 4, no. 3, pp. 247-258.

Rational (2002). http://www.rational.com/products/rose/index.jsp

Riel, A., (1994). "Object-Oriented Design Heuristics", Addison-Wesley.

Robbins, J. (1998) "Design Critiquing Systems", Technical Report UCI-98-41, University of California, Irvine

Robbins, J. and Redmiles D., (1998) "Software Architecture Critics in the Argo Design Environment", Knowledge-Based Systems, 11(1)

SEI (2002). "A Framework for Software Product Line Practice - Version 3.0", http://www.sei.cmu.edu/plp/framework.html

SPC (1993). Reuse-Driven Software Processes Guidebook, Version 02.00.03. Technical Report SPC-92019-CMC, Software Productivity Consortium, November

SPCSC (1993). Software Productivity Consortium Service Corporation. Reuse Adoption Guidebook, Version 02.00.03, November

Sun Microsystems, Inc (2001). Jini™ Technology Core Platform Specification, http://wwws.sun.com/software/jini/specs/jini1.2html/core-title.html, December

SWI-Prolog (1987), http://www.swi-prolog.org/

Tarr, P., Ossher, H., Harrison, W., and Sutton, S., (1999). N Degrees of Separation: Multi-Dimensional Separation of Concerns. Proceedings of the International Conference on Software Engineering (ICSE'99), May

Terry, A., T. Dabija, T. Barnes, and A. Teklemariam (1995) "DADSE 2.3 User Manual", Teknowledge Federal Systems

Tracz, W. and L. Coglianese (1995) "DOMAIN (DOmain Model All Integrated): A DSSA Domain Analysis Tool", Technical Report ADAGE-LOR-94-13

Weiss D., Lai R. (1999). "Software Product Line Engineering: A Family Based Software Development Process", Addison-Wesley

W3C, "Extensible Markup Language (XML) 1.0", W3C Recommendation, REC-xml-19980210, URL: http://www.w3.org/TR/REC-xml, 1998